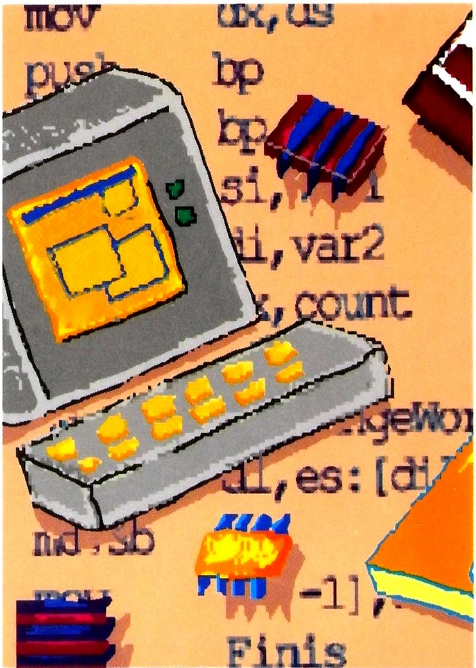
TURBO DEBUGGER®

USER'S GUIDE



BORLAND

TURBO DEBUGGER®

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book,* with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated).

The sample programs included on the Turbo Debugger diskettes provide a demonstration of how to use the various features of Turbo Debugger. They are intended for educational purposes only. Borland International grants you (the registered owner of Turbo Debugger) the right to edit or modify these sample programs for your own use, but you may not give away or sell them, alone or as part of any program, in executable, object or source code form. You may, however, incorporate miscellaneous sample program routines into your programs, as long as your resulting programs do not substantially duplicate all or part of a sample program in appearance or functionality and all copies of such programs bear a copyright notice in a manner prescribed by the copyright laws.

LIMITED WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically **disclaims** all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. **In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.**

GOVERNING LAW

This statement shall be construed, interpreted, and governed by the laws of the state of California. Use, duplication, or disclosure by the U.S. Government of the computer software and documentation in this package shall be subject to the restricted rights under DFARS 52.227-7013 applicable to commercial computer software.

1 TDHEAD 890318

Turbo Debugger®

ユーザーズガイド

バージョン1.0

(NEC PC-9801シリーズ)
(IBM PC, XT, AT, PS/2および完全な互換機)

Copyright © 1988
All Right Reserved

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

編訳 (株)ボーランド ジャパン 1989

本マニュアルの一部または全部を許可なく複製または転載することはできません。

目　次

はじめに……1

Turbo Debuggerを使用するために必要なもの……2

用語に関する注意……3

このマニュアルの内容……4

製品に関するお問い合わせ……8

第1章　使い始める前に……9

Turbo Debuggerのパッケージ……10

マスターディスクの内容……11

Turbo Debuggerのインストール……12

フロッピィディスクシステムで使用するためのセットアップ……12

2台のフロッピィディスクでTurbo Debuggerを使用する方法……12

ハードディスクシステムで使用するためのセットアップ……13

ハードウェアデバッガ……14

次にどこに進めばよいか……15

Turbo言語を学び始めたばかりの人……15

Turbo言語をすでに使っている人……15

第2章　デバッグとTurbo Debugger……17

デバッグとは何か？……18

バグがあるか？……18

どこにあるか？……18

原因は何か？……19

バグの修正……19

Turbo Debuggerができること……20

Turbo Debuggerにできないこと……21

Turbo Debuggerのユーザインターフェース……22

Turbo Debuggerの特長……23

メインメニューとプルダウンメニュー……23

状況感知機能……24

ローカルメニュー……………………………………………………25
プルダウン(グローバル)メニュー……………………………………25
ポップアップ(ローカル)メニュー……………………………………26
ヒストリリスト……………………………………………………27
マクロの作成………………………………………………………29
ウィンドウ…………………………………………………………29
Viewメニューから開くことができるウィンドウ ……………………30
Inspectorウィンドウ ………………………………………………34
アクティブウィンドウ………………………………………………34
ウィンドウ間の移動…………………………………………………35
ウィンドウの再設定と保存…………………………………………36
ヘルプの呼び出し……………………………………………………37
オンラインヘルプ……………………………………………………37
最下行………………………………………………………………38

第3章　チュートリアル ……………………………………………41
例題プログラム………………………………………………………42
Turbo Debuggerを使う ……………………………………………44
ヘルプ行……………………………………………………………44
ウィンドウ…………………………………………………………44
Turbo Debuggerの使い方（Cプログラムのデバッグ） ……………46
ブレークポイントを設定する………………………………………47
Watchesウィンドウ…………………………………………………48
Cの単純なデータを調べる …………………………………………49
Cの複合データを調べる ……………………………………………51
Cのデータの値を変更する …………………………………………52
Turbo Debuggerの使い方（Pascalプログラムのデバッグ） ………54
ブレークポイントを設定する………………………………………55
Watchesウィンドウ…………………………………………………56
Pascalの単純なデータを調べる ……………………………………57
Pascalの複合データを調べる ………………………………………58
Pascalのデータの値を変更する ……………………………………59

第4章　Turbo Debuggerの起動……………………………………61
プログラムをデバッグできるように準備する………………………62

Turbo Cプログラムの準備……62
Turbo Pascalプログラムの準備……62
Turbo Assemblerプログラムの準備……63
マイクロソフト社の言語で書かれたプログラムの準備……63
Turbo Debuggerの起動……64
コマンドラインオプション……66
-cオプション……66
-dオプション……66
-do……66
-dp……67
-ds……67
-gオプション(PC-9801のみ)……67
-gN……67
-hと-?オプション……67
-iオプション……68
-lオプション……68
-mオプション……68
-rオプション……68
-rpN……68
-rsN……69
-sオプション……69
-sc……69
-sd……69
-vオプション(IBM PCのみ)……70
-vg……70
-vn……70
コンフィギュレーションファイル……71
Optionsメニュー……72
Languageコマンド……72
Macrosコマンド……72
Create……73
Stop recording……73
Remove……73
Delete all……73
Environmentコマンド……73

Integer format……74
Display swapping……74
Screen size……74
Tab size……75
Path for sourceコマンド……75
Argumentsコマンド……75
Save optionsコマンド……75
Restore optionsコマンド……76
Hardware statusコマンド（PC-9801のみ）……76
Visual user G-page……76
Active user G-page……76
Graphics switch……76
Cursor switch……77
EMS/VRAM switch……77
Turbo Debuggerの内部からDOSのコマンドを実行する方法……78
DOSに戻る方法……78

第５章　プログラムの実行を制御する……79
プログラムの現在の状態を調べる方法……80
Variablesウィンドウ……80
Globalペインのローカルメニュー……82
Inspect……82
Change……83
Staticペインのローカルメニュー……83
Inspect……84
Change……84
Stackウィンドウ……85
Stackウィンドウのローカルメニュー……85
Inspect……86
Locals……86
Originローカルメニューコマンド……87
Get infoコマンド……87
Runメニュー……90
Run［F9］……90
Program reset［CTRL-F2］……91

Go to cursor [F4] ……91
Trace into [F7] ……91
Step over [F8] ……91
Execute to [GRPH-F9] ……92
Until return [GRPH-F8] ……92
Animate [GRPH-F4] ……92
Instruction trace [GRPH-F7] ……93
プログラムの実行を停止させる方法……94
CTRL-STOP (Ctrl-Break) ……94
プログラムを終了させる……95
デバッグセッションを再開する……96
プログラムを再ロードする……96
キーストロークの記録と再生……96
デバッグのために新しいプログラムをロードする……98
プログラムの引数を変更する……98

第6章 データを調べる，変更する ……99
Dataメニュー……100
Inspect……100
Evaluate/modify ……101
Cプログラマへの注意 ……102
Watch ……102
Function return……102
ソースファイルの中のデータを指定する ……103
Watchesウィンドウ ……104
Watchesウィンドウのローカルメニュー ……106
Watch ……106
Edit ……106
Remove ……107
Delete all……107
Inspect……107
Change……107
Inspectorウィンドウ……108
CのデータのInspectorウィンドウ ……108
スカラー ……108

ポインタ ……109
配列 ……111
構造体と共用体 ……111
関数 ……113
PascalのデータのInspectorウィンドウ ……114
スカラー ……114
ポインタ ……115
配列 ……116
レコード ……117
手続きと関数 ……117
アセンブラのデータのInspectorウィンドウ ……118
スカラー ……118
ポインタ ……119
配列 ……121
構造体と共用体 ……121
Inspectorウィンドウのローカルメニュー ……123
Range ……123
Change ……124
Inspect ……124
Descend ……124
New expression ……125

第7章　ブレークポイント ……127
Breakpointsメニュー ……129
Toggle ……129
At... ……129
Changed memory global... ……130
Expression true global... ……130
Delete all ……130
ブレークポイント式に適用されるスコープ（適用範囲） ……131
Breakpointsウィンドウ ……132
Breakpointsウィンドウのローカルメニュー ……133
Set action ……133
Break ……134
Log ……134

Execute ····· 134
Condition ····· 135
Always ····· 135
Changed memory... ····· 135
Expression true ····· 136
Hardware ····· 136
Pass count... ····· 136
Enable/disable ····· 137
Add... ····· 137
Global ····· 137
Remove ····· 138
Delete all ····· 138
Inspect ····· 138
Logウィンドウ ····· 139
Logウィンドウのローカルメニュー ····· 140
Open log file ····· 141
Close log file ····· 141
Logging ····· 141
Add comment ····· 141
Erase log ····· 141
単純ブレークポイント ····· 142
条件ブレークポイントとパスカウント ····· 143
グローバルブレークポイント ····· 144
データが変化したときにブレークする ····· 145
変数の値をログに書き込む ····· 146
式を評価する ····· 147

第8章 ファイルを調べる，変更する ····· 149
プログラムのソースファイルを調べる ····· 150
Moduleウィンドウ ····· 150
Moduleウィンドウのローカルメニュー ····· 153
Inspect ····· 153
Watch ····· 154
Module ····· 154
File ····· 154

Previous ……155
Line ……155
Search ……155
Next ……155
Origin ……156
Goto ……156
Edit ……156
他のディスクファイルを調べる ……157
Fileウィンドウ ……157
Fileウィンドウのローカルメニュー ……158
Goto ……159
Search ……159
Next ……160
Display as ……160
File ……161
Edit ……161

第9章　式 ……163
式を評価するための言語の選択 ……164
コードアドレス，データアドレス，行番号 ……164
カレントスコープ外のシンボルへのアクセス ……165
スコープをオーバーライドするための書式（文法） ……165
式の評価のために使用される暗黙のスコープ ……167
バイトリスト ……168
Cの式 ……169
Cのシンボル ……169
Cのレジスタ疑似変数 ……169
Cの定数と数の表記 ……170
Cの文字列とエスケープシーケンス ……171
Cの演算子と演算子の優先順位 ……172
式の中でCの関数を呼び出す ……173
副作用のあるCの式 ……173
Cのキーワードとキャスト（強制型変換） ……174
Pascalの式 ……176
Pascalのシンボル ……176

Pascalの定数と表記 ……176
Pascalの文字列 ……177
Pascalの演算子 ……177
Pascalの関数と手続きの呼び出し ……178
アセンブラの式 ……179
アセンブラのシンボル ……179
アセンブラの定数 ……179
アセンブラの演算子 ……180
書式制御 ……181

第10章　アセンブラレベルのデバッグ ……183
ソースデバッグでは十分でないとき ……184
CPUウィンドウ ……185
Codeペイン ……188
逆アセンブラ ……188
Codeペインのローカルメニュー ……189
Goto ……190
Origin ……190
Follow ……191
Caller ……191
Previous ……191
Search ……191
Mixed ……192
New CS:IP ……192
Assemble ……193
I/O ……193
In byte ……194
Out byte ……194
Read word ……194
Write word ……194
Registerペインのローカルメニュー ……194
Increment ……195
Decrement ……195
Zero ……195
Change ……195

Registers 32-bit ……196
Flagsペインのローカルメニュー ……196
Toggle ……197
Dataペイン ……198
Dataペインのローカルメニュー ……198
Goto ……199
Search ……199
Next ……199
Change ……200
Follow ……200
Long follow ……200
Previous ……200
Display as ……200
Byte ……201
Word ……201
Long ……201
Comp ……202
Float ……202
Real ……202
Double ……202
Extended ……202
Block ……202
Clear ……203
Move ……203
Set ……203
Read ……204
Write ……204
Stackペインのローカルメニュー ……204
Goto ……205
Origin ……205
Follow ……206
Previous ……206
Change ……206
アセンブラ ……207
オペランドのアドレスサイズのオーバーライド ……207

メモリオペランドと即値オペランド …… 207
オペランドのデータサイズのオーバーライド …… 208
ストリング命令 …… 208
Dumpウィンドウ …… 209
Registersウィンドウ …… 209
Turbo Cのコード生成 …… 210

第11章　80x87コプロセッサチップとエミュレータ …… 211
80x87チップ対エミュレータ …… 212
Numeric Processorウィンドウ …… 213
80ビット浮動小数点レジスタ …… 214
ステータスビット …… 214
コントロールビット …… 214
Registerペインのローカルメニュー …… 215
Zero …… 216
Empty …… 216
Change …… 216
Statusペインのローカルメニュー …… 217
Toggle …… 217
Controlペインのローカルメニュー …… 218
Toggle …… 218

第12章　コマンドリファレンス …… 219
ホットキー …… 220
メインメニューバーから選択できるコマンド …… 222
Fileメニュー …… 222
Viewメニュー …… 222
Runメニュー …… 223
Breakpointsメニュー …… 223
Dataメニュー …… 223
Windowメニュー …… 224
Optionsメニュー …… 224
Localメニューコマンド …… 226
Breakpointsウィンドウのローカルメニュー …… 226
CPUウィンドウのローカルメニュー …… 228

Codeペインのローカルメニュー……228
Dataペインのローカルメニュー……229
Stackペインのローカルメニュー……229
Registerペインのローカルメニュー……230
Flagsペインのローカルメニュー……230
Fileウィンドウのローカルメニュー……230
Logウィンドウメニューのローカルメニュー……231
Moduleウィンドウのローカルメニュー……231
Numeric Processorウィンドウのローカルメニュー……232
Registerペインのローカルメニュー……232
Statusペインのローカルメニュー……232
Controlペインのローカルメニュー……232
Stackウィンドウのローカルメニュー……233
Variableウィンドウのローカルメニュー……233
Global Symbolペインのローカルメニュー……233
Static Symbolペインのローカルメニュー……233
Watchesウィンドウのローカルメニュー……234
Inspectorウィンドウのローカルメニュー……234
Textペイン……235
Listペイン……237
プロンプトボックスの中にいるときのコマンド……238
ウィンドウ操作コマンド……239
検索のワイルドカード……240
ファイルリスト……241
メニュー構成……242

第13章　デバッグの進め方……247
プログラムが正しく動作しないとき……248
デバッグスタイル……249
プログラムの全体的な動作を調べる……249
プログラムの各部分からデバッグする……249
バグの種類……251
Cに特有のバグ……251
初期化されていない自動変数を使う……251
代入(=)と比較(==)を混同する……252

演算子の優先順位を間違える ……252
ポインタの演算の間違い ……252
符号が拡張されることを忘れる ……253
桁の切り詰めを忘れる ……253
余分なセミコロン ……254
副作用のあるマクロ ……254
自動変数名の重複 ……255
自動変数の使い方が間違っている ……255
関数が値を返さない場合がある ……255
break文の使い方の間違い ……256
何もしないコード ……256
一般的なバグ ……257
隠れた副作用を忘れる ……257
別のところで初期化されていると思い込む ……257
不要になったときにメモリ割り当てを解放しない ……258
フェンスポスト(柵の支柱)エラー ……258
Pascalに特有のバグ ……259
初期化されていない変数を使う ……259
初期化していないポインタを使用する ……259
スコープを間違える ……260
余分なセミコロン ……262
関数名に値が代入されない場合がある ……262
word型またはbyte型の変数をデクリメントする ……263
境界や特殊な場合を忘れる ……264
範囲エラー ……264
アセンブリ言語に特有のバグ ……265
DOSに戻るのを忘れる ……266
RET命令を忘れる ……266
生成されるリターンのタイプが間違っている ……266
オペランドの順番が逆になっている ……266
スタックスペースを確保するのを忘れる,
または確保したスペースが小さすぎる ……267
必要なレジスタを破壊するサブルーチンを呼び出す ……267
条件処理でジャンプ命令またはジャンプ条件を間違える ……267
REPつきのストリング命令で文字列をオーバーランする ……268

CXがゼロのとき何もしないことを忘れて
セグメント全体がカバーされると思い込む ……268
ディレクションフラグの設定が間違っている ……268
文字列比較に間違った比較命令または間違ったプレフィクスを
使用する ……269
ストリング命令のデフォルトセグメントを忘れる ……269
BYTE操作からWORD操作への変換に誤りがある ……269
複数のプレフィクスを使用する ……270
ストリング命令に対するオペランドの指定が有効と思い込む ……270
乗算によりレジスタを破壊する ……270
ストリング命令がいくつかのレジスタの値を変えるのを忘れる ……270
フラグを変化させない命令であるのに変化させると勘違いする ……270
フラグを利用する命令がフラグがセットされる場所から
離れすぎている ……271
メモリ参照オペランドと即値オペランドを間違える ……271
セグメントの終わりから先頭に戻る ……271
割り込みハンドラですべてのデータを退避していない ……271
オペランドおよびデータテーブルに
グループオーバーライドを忘れる ……272
動作テスト ……273
範囲の境界および限界をテストする ……273
間違ったデータを入力してみる ……273
何も入力しない ……273
プログラム設計の一部としてのデバッグ ……274
サンプルデバッグセッション ……275
Cのデバッグセッション ……276
間違いを探す ……276
どの問題から取りかかるかを決める ……277
Turbo Debuggerを起動する ……277
Inspectコマンドで調べる ……278
バグの発見と修正 ……279
Pascalのデバッグセッション ……281
間違いを探す ……281
どの問題から取りかかるかを決める ……282
Turbo Debuggerを起動する ……283

Inspectコマンドで調べる……285
Watchコマンドで調べる……286
残りのもうひとつのバグ……287

第14章　80386プロセッサシステムでの仮想デバッグ……289
仮想デバッグのために必要なもの……290
仮想デバッガのデバイスドライバのインストール……290
仮想デバッガの起動……290
通常デバッグと仮想デバッグの相違……293
TD386のエラーメッセージ……294
TDH386.SYSエラーメッセージ……296

付録A　コマンドラインオプション……297

付録B　Turbo Debuggerのユーティリティ……299
シンボルテーブルコンバータ：(CodeView→Turbo Debugger)……300
DOSから起動する……300
エラーメッセージ……300
TDRFユーティリティ……303
DOSのコマンドラインからTRDFを起動する……303
TDRFのコマンドライン……304
TDRFのコマンド……305
TDRFのエラーメッセージ……308
シンボルテーブル削除ユーティリティ：TDSTRIP……312
TDSTRIPのコマンドライン……312
TDSTRIPのエラーメッセージ……313
TDMAPユーティリティ……316
TDPACKユーティリティ……317
TDUMPユーティリティ……318
TDUMPのコマンドライン……318
TDUMPのオプション……318
-aと-a7オプション……319
-e, -el, -erオプション……319
-hオプション……319
-lオプション……320

-oオプション …… 320
-vオプション …… 320

付録C　技術ノート …… 321
ロードアドレスと空きメモリの変化 …… 322
システムの破壊 …… 322
DOS内部のトレースとプロセスIDの切り替え …… 323
8087/80287数値演算コプロセッサとエミュレータ …… 323
割り込み …… 324
INT 1とINT 3を使うプログラムのデバッグ …… 325
ディスプレイモードの退避と切り替え …… 325
メモリ上の配置 …… 325
EMSのサポート …… 326
割り込みベクターの退避と回復 …… 326

付録D　組み込みアセンブラのキーワード …… 327

付録E　Turbo Debuggerのカスタマイズ …… 331
TDINSTの起動とオプションの設定方法 …… 332
画面の表示色の設定 …… 332
ディスプレイパラメータの設定 …… 335
Beginning displayオプション …… 335
User screen updatingオプション …… 336
Display swappingオプション …… 337
Integer formatオプション …… 337
Log list lengthオプション …… 338
Tab column widthオプション …… 338
Maximum tiled watchオプション …… 338
Screens linesオプション …… 338
Fast screen updateオプション（IBM PCのみ） …… 338
Permit 43/50 line modeオプション（IBM PCのみ） …… 339
Complete graphics saveオプション（IBM PCのみ） …… 339
Turbo Debuggerのオプションの設定 …… 340
Editorオプション …… 340
Source directoriesオプション …… 341

Turbo directoryオプション ……341
Keysオプション ……341
Promptingオプション ……342
Remote debuggingオプション ……342
OS shell swap sizeオプション ……343
Languageオプション ……343
Ignore caseオプション ……344
Change process IDオプション ……344
Use expanded memoryオプション (IBM PCのみ) ……344
Mode for hardwareメニュー (PC-9801) ……345
Screen modeオプション ……345
Buffer of symbolオプション ……346
Graphics switchesオプション ……346
Mode for displayメニュー (IBM PC) ……347
コマンドラインオプションとTDINSTのメニューオプションの対応 ……348
設定の保存とTDINSTの終了 ……349

付録F　ハードウェアデバッガインターフェース ……351
80386ハードウェアデバイスドライバ ……352
ハードウェアブレークポイントの設定方法 ……353
TDH386.SYSにより設定できるハードウェア
ブレークポイントのマッチング条件 ……354
BreakpointsウィンドウのHardwareメニュー ……355
ハードウェアデバッガの概要 ……358
デバイスドライバインターフェース ……359
INIT　コマンドコード＝0 ……359
READ　コマンドコード＝4 ……359
READNOWAIT　コマンドコード＝5 ……359
READSTATUS　コマンドコード＝6 ……360
READFLUSH　コマンドコード＝7 ……360
WRITE　コマンドコード＝8 ……360
WRITEVERIFY　コマンドコード＝9 ……360
WRITESTATUS　コマンドコード＝10 ……360
WRITEFLUSH　コマンドコード＝11 ……360
デバイスドライバに送られるコマンドブロック ……361

Install vectors（コード0）……361
Set a hardware breakpoint（コード4）……362
Clear a hardware breakpoint（コード5）……364
Set I/O board base address（コード9）……364
デバイスドライバにより返されるステータスブロック……365
Get hardware capabilities（コード1）……365
Set a hardware breakpoint（コード4）……367
デバイスドライバからのTurbo Debuggerの呼び出し……368

付録G　リモートデバッグ……369
リモートデバッグのためのハードウェアのセットアップ……370
リモートシステムのソフトウェアの準備……371
TDREMOTEの起動……371
Turbo Debuggerの起動……372
リモートシステムへのプログラムのロード……373
TDREMOTEのコマンドラインオプション……373
リモートデバッグセッション……375
TDREMOTEのメッセージ……375
システム間のリンクに問題があるとき……379

付録H　プロンプトとエラーメッセージ……381
プロンプト……382
エラーメッセージ……391
致命的エラー……391
エラーメッセージ……393
情報メッセージ……415

付録I　Turbo Debuggerをいろいろな言語に使用する……417
Turbo Cのための助言……418
コンパイラによるコードの最適化……418
ポインタデータへのアクセス……418
複雑な式の部分式の値を調べる……419
Turbo Assemblerのための助言……420
データの16進ダンプ……420
ソースレベルのデバッグ……420

レジスタの内容を調べる，変更する ……420
Turbo Pascalのための助言 ……422
初期化コードを1行ずつ実行する ……422
終了手続きを1行ずつ実行する ……422
定数 ……422
文字列演算および集合演算の中間結果の一時格納スペース ……423
巧妙な型キャスト ……423
PascalでCPUウィンドウを使うときのアドバイス ……424

付録J　用語集 ……425

索引 ……431

はじめに

Turbo Debuggerは，ボーランド社のTurbo C，Turbo Pascal，Turbo Assemblerでプログラムを書いている人，他のコンパイラやアセンブラを使用していて，より強力なデバッグ環境を望んでいる人を対象とする，先進的なソースレベルデバッガです。

オーバーラップ可能な複数のウィンドウと，プルダウンメニューとポップアップメニューの巧みな組み合わせが，応答の速い会話型のデバッグ環境を提供します。またデバッグセッションのすべての局面で，状況に応じた適切な情報を表示するヘルプシステムが援助の手を差し伸べます。

Turbo Debuggerの主な特長は次のとおりです。

■EMS(Expanded Memory Specification)を使用する大規模なプログラムのデバッグ
■リモートシステム上での大きなプログラムのデバッグ
■すべてのCの式を評価可能
■すべてのPascalの式を評価可能
■画面表示のレイアウトを再設定可能
■必要なときにいつでもアセンブラ/CPUにアクセス可能
■強力なブレークポイントとログ機能
■ヒストリリストとキーストロークマクロによるキー入力の省力化
■80386や他メーカーのデバッグ用ハードウェアのサポート

Turbo Debuggerを使用するために必要なもの

Turbo DebuggerはNEC PC-9801シリーズのコンピュータ（あるいはIBM PC, XT, AT, PS/2, およびそれらの完全互換機）上で動作します。MS-DOS（またはPC-DOS）バージョン2.1あるいはそれ以降のバージョンと，384Kバイト以上のRAMが必要です。80桁のディスプレイであれば，カラーでもモノクロームでも動作します。2台のディスクドライブでも良好に動作しますが，ハードディスクの使用をお奨めします。

Turbo Debuggerは，8087数値演算コプロセッサチップは必要としません。

Turbo Debuggerをボーランド社の他の製品とともに使用するには，Turbo Pascalのバージョン5.0またはそれ以降のバージョン，Turbo Cのバージョン2.0またはそれ以降のバージョン，Turbo Assemblerのバージョン1.0またはそれ以降のバージョンを使用する必要があります。Turbo Debuggerでデバッグするには，その前に，すべてのデバッグ情報の生成をオンにして，ソースコードを実行可能ファイル(.EXEファイル)にコンパイルする必要があります。

Turbo Debuggerを使うときには，.EXEファイルと元のソースファイルの両方が必要です。Turbo Debuggerは，ソースファイルを見つけるために，まずコンパイラがソースファイルを見つけ出したディレクトリを探します。2番目にOptionsメニューのPath for sourceコマンドで指定されたディレクトリ，3番目にカレントディレクトリ，そして最後に.EXEファイルの入っているディレクトリを探します。

用語に関する注意

このマニュアルでは，説明の便宜上，次の用語を通常使用されるより多少広い意味で使っています。

モジュール　Cやアセンブラでモジュールと呼ばれるものを指すだけでなく，Pascalで**ユニット**と呼ばれるものも含みます。

関数　Cの関数と，Pascalでサブプログラム（またはサブルーチン）と呼ばれるものを指します。Pascalのサブプログラムには，**関数**と**手続き**が含まれます。Cの関数は，Pascalの関数と同じように値を返すこともできますし，Pascalの手続きと同じように値を返さなくてもかまいません（Cの関数が値を返さないときは，その関数は**void関数**と呼ばれます）。このマニュアルでは説明を簡潔にするために，特定の言語のための説明を除いて，〝**関数**〟をCの関数，Pascalの関数，およびPascalの手続きを含む広い意味で使っています。

引数　**パラメータ**と同じ意味に使用します。関数や手続きの引数やコマンドライン引数の〝**引数**〟も，同じ意味で使用しています。

このマニュアルの内容

このマニュアルの各章と付録の概要を以下に示します。

第1章　使い始める前に

マスターディスクの内容と，Turbo Debuggerをシステムにロードする方法について説明します。また習熟度に応じて，次にどの章に進むべきかをアドバイスします。

第2章　デバッグとTurbo Debugger

Turbo Debuggerのユーザインターフェース，メニュー，ウィンドウについて説明します。またプロンプトやエラーメッセージにどう対処すればよいかを示します。

第3章　チュートリアル

PascalまたはCのサンプルプログラムのうちどちらか好きな方を使って，デバッグのやり方を手ほどきします。Turbo Debuggerの強力な機能の威力を示します。

第4章　Turbo Debuggerの起動

DOSのプロンプトからデバッガを起動する方法，コマンドラインオプションの意味と指定の方法，通常使用する環境設定をコンフィギュレーションファイルに保存する方法を説明します。

第5章　プログラムの実行を制御する

デバッグするプログラムの実行を開始したり停止したりする様々な方法，デバッグセッションの再開のしかた，最後のセッションをもう一度繰り返す方法について説明します。

第6章　プログラムデータを調べる，変更する

プログラムの中のデータを調べたり変更したりするための，強力な機能について説明します。

第7章　ブレークポイント

ブレークポイントのアクションについて説明します。また無条件ブレークと条件ブレークについて説明します。さらにブレークポイントのアクションがトリガされたときにできることを説明します。

第8章　ソースファイルを調べる，変更する

プログラムのソースファイルを調べたり変更したりする方法を説明します。また任意のディスクファイルを，テキスト形式またはバイナリデータ形式で，調べたり変更したりする方法について説明します。

第9章　式

Turbo DebuggerがサポートするC，Pascal，およびアセンブラの式の規則と，式の値の表示形式を変更するために使用する書式制御文字について説明します。

第10章　アセンブラレベルのデバッグ

メモリの内容を生の16進数データとして見たり変更したりする方法，組み込みのアセンブラおよび逆アセンブラの使い方，CPUのレジスタやフラグを調べたり変更したりする方法について説明します。

第11章　8087/80287数値演算コプロセッサチップとエミュレータ

浮動小数点数値演算用ハードウェアまたはエミュレータの内容を調べたり変更したりするための方法を説明します。

第12章　コマンドレファレンス

メインメニューコマンド，プルダウンメニューコマンド，ローカルメニューコマンドを表にまとめてあります。

第13章　デバッグのやり方

プログラムを効率的にデバッグするための方策を説明します。

第14章　80386プロセッサシステムでの仮想デバッグ

80386コンピュータの大きなメモリと強力な機能を利用して，デバッグするプログラムに，640Kより下位の全アドレス空間を使用してデバッグすることができるデバッグ方法について説明します。

付録A　コマンドラインオプション

第4章で説明されているコマンドラインオプションをまとめてあります。

付録B　Turbo Debuggerのユーティリティ

Turbo Debuggerとともに提供されるユーティリティの使い方を説明します。ユーティリ

ティプログラムには,マイクロソフト社のコンパイラで開発されたプログラムをTurbo Debuggerでデバッグできるようにするためのプログラムをはじめ,.EXEファイルに付加されるデバッグ情報に影響を与えるいくつかのプログラムがあります。またファイルの構成要素を表示させることができるTDUMPという名前のユーティリティもあります。

付録C　技術ノート

経験のあるプログラマを対象としています。Turbo Debuggerのインプリメンテーションの詳細を説明します。Turbo Debuggerとデバッグしているプログラムや DOSとの相互作用がわかります。

付録D　組み込みアセンブラのキーワード

Cのプログラムの中にアセンブリ命令を入れるときに使用される,8086/80286の命令のニーモニックと命令以外の特別な語を表にしてあります。

付録E　Turbo Debuggerのカスタマイズ

インストールプログラム(TDINST)を使って，画面の表示色を好みに合せて再設定したり，デフォルトのオプションを変更したりする方法について説明します。

付録F　ハードウェアデバッガインターフェース

Turbo Debuggerとともに提供されるデバイスドライバのインストールの手順と,特殊なハードウェアデバッガをTurbo Debuggerとともに使用するためのデバイスドライバの書き方について説明します。

付録G　リモートデバッグ

コンピュータに接続された第2のコンピュータを使って,非常に大きなプログラムをデバッグする方法を説明します。

付録H　プロンプトとエラーメッセージ

画面に表示される可能性のある全てのプロンプトとエラーメッセージと，そのメッセージが表示されたときの対処方法がまとめてあります。

付録I　Turbo Debuggerをいろいろな言語に使用する

他社のC, アセンブラ, Pascalなどで書かれたプログラムをデバッグしているときに役に立つ助言が載っています。

付録J　用語集

このマニュアルで使用されている専門用語の簡潔な定義をまとめてあります。

製品に関するお問い合わせ

ソフトウェアおよびマニュアルに関するご意見・ご質問は，下記代理店の**MSAカスタマサポートセンター**まで，お電話，書面またはファクシミリにてお問い合わせください。なお，お電話での受付時間は平日の9:00am～12:00amおよび1:00pm～5:00pmとなっています（土曜，日曜，祝日は休業となりますのでご了承ください）。

〒107 東京都港区南青山7-8-1 小田急南青山ビル
(株)マイクロソフトウェア アソシエイツ
カスタマーサポートセンター
電話　03-486-1403
FAX　03-486-8905

お問い合わせの際には下記の事項をお伝えください。

■製品名とバージョーン番号（Turbo Debugger 1.0など）
■製品のシリアル番号（マスターディスクに記載されているもの）
■コンピュータ名（PC-9801VXなど）
■オペレーティングシステムとバージョン番号（MS-DOS 3.3など）

なお，書面にてお問い合わせの際には，CONFIG.SYSおよびAUTOEXEC.BATの内容，簡潔なプログラムリストを添付されるようお願い致します。

第1章

使い始める前に

Turbo Debuggerのパッケージ

Turbo Debuggerのパッケージには，1枚のマスターディスクとTurbo Debuggerユーザーズガイド（この本）が入っています。マスターディスクには，Turbo C，Turbo Assembler，Turbo Pascalのプログラム，およびマイクロソフト社のコンパイラにより作成されたプログラムをデバッグするために必要なプログラム，ファイル，およびユーティリティが含まれています。Turbo Debuggerのパッケージには，その他に，このマニュアルには記載されていない事項に関するドキュメントファイルが含まれています。

ユーザーズガイドには，Turbo Debuggerの機能の説明と，コマンドリファレンスが記載されています。

Turbo Debuggerを使い始める前にマスターディスクからコピーを作り，マスターディスクは安全な場所に保管しておいてください。マスターディスクはバックアップのためだけに使用するようにし，普段はコピーしたディスクからTurbo Debuggerを起動するようにしてください。常用ディスクに何か問題が発生した場合には，マスターディスクからコピーし直してください。

マスターディスクの内容

Turbo Debuggerをあなたのシステムにインストールするときは，マスターディスクから常用のフロッピィディスクまたはハードディスクにファイルをコピーしてください。マスターディスクはコピープロテクトされていませんし，インストールプログラムを起動する必要もありません。

マスターディスクには，以下のファイルが入っています。

TD.EXE	Turbo Debugger
TDINST.EXE	Turbo Debuggerインストールプログラム
TDHELP.TDH	Turbo Debuggerヘルプファイル
README.COM	ドキュメントファイルREADMEを読むためのプログラム
README	発売直前の情報
TDH386.SYS	80386ハードウェアデバイスドライバ
TD386.EXE	仮想デバッグのために使用するプログラム
TDRF.EXE	リモートデバッグ時の2つのシステム間のファイル転送ユーティリティ
TDREMOTE.EXE	リモートデバッグのために使用するプログラム
TDSTRIP.EXE	シンボルテーブルを削除するためのユーティリティ
TDUMP.EXE	モジュール逆アセンブルユーティリティ
TDCONVERT.EXE	CodeView形式のプログラムをTurbo形式に変換するためのユーティリティ
TDMAP	.EXEファイルに.MAPファイル情報を付加するためのユーティリティ
TDPACK	.EXEファイルの中のデバッグ情報のサイズを小さくするためのユーティリティ
TCDEMO.*	第3章のチュートリアルで使用されるCの例題プログラム
TCDEMOB.*	第13章の説明で使用されるバグのあるCの例題プログラム
TPDEMO.*	第3章のチュートリアルで使用されるPascalの例題プログラム
TPDEMOB.*	第13章の説明で使用されるバグのあるPascalの例題プログラム

Turbo Debuggerのインストール

マスターディスクのすべてのファイルを，常用のフロッピィディスクまたはハードディスクにコピーしてください。すべてのファイルをコピーするにはスペースが足りないときでも，TD.EXEとTD.OVLだけはコピーしなければなりません。

TD.OVLは，メニューシステムが入っているファイルです。Turbo Debuggerがメニューを表示するときは，このファイルが必要です。Turbo Debuggerは，起動時にこのファイルが見つからないと，そのことを知らせるエラーメッセージを表示します。またTurbo Debuggerをディスクから起動しているときは，TD.OVLの入っているディスクを挿入するように求めるメッセージを表示します。

フロッピィディスクシステムで使用するためのセットアップ

2台のフロッピィディスクでTurbo Debuggerを使用する方法

フロッピィディスクドライブが2台のシステムでTurbo Debuggerを使用するには，2枚のフロッピィディスクを作成する必要があります。デバッガ(TD.EXE)を含むプログラムディスクと作業ディスクです。

プログラムディスクには，マスターディスクからすべてのファイルをコピーします。

作業ディスクは，デバッグしようとする実行可能なプログラムと，そのソースファイルを入れるために使用します。

フロッピィディスクドライブが2台のシステムからTurbo Debuggerを起動するには，プログラムディスクをドライブAに，作業ディスクをドライブBに挿入します。カレントドライブがドライブAであることを確認してから，〝TD〞と入力して**リターン**を押します。次に**File/Change dir**コマンドを選択し〝B：〞と入力して，カレントドライブをドライブBに変更します。最後に**File/Load**コマンドを選択して，デバッグするプログラムをロードします。

ハードディスクシステムで使用するためのセットアップ

ハードディスクのついたシステムで使用すると，デバッグの能率が大幅に向上します。

既存のディレクトリの中にデバッガをセットアップしたいときは，まずそのディレクトリに移り，つぎにマスターディスクのすべてのファイルをそのディレクトリにコピーします。たとえば，Turbo Cを使っていて，そのディレクトリの中でデバッガを使いたいときは，次のように入力します。

```
A>copy b:*.* a:¥TC
```

*TC*はTurbo Cの入っているディレクトリとします。Turbo Pascalのユーザは，同じようにして，Turbo Debuggerのマスターディスクの内容をPascalのディレクトリにコピーしてください。

Turbo Debuggerのために新しいディレクトリを作成し，そのディレクトリの中にマスターディスクからすべてのファイルをコピーすることもできます。コピーが済んだら，デバッガをコピーしたディレクトリに移り，デバッガを起動します。他のディレクトリからTurbo Debuggerを起動するときは，PATH環境変数にデバッガディレクトリを含めます。このようにすると，システムのどこからでもデバッガ起動することができます。Turbo Debuggerは，PATH環境変数を使って，ヘルプファイルとコンフィギュレーションファイルを見つけだします。

ハードウェアデバッガ

80386システムを使用されている方は，Turbo DebuggerとともにTDH386.SYSデバイスドライバをインストールすることができます。このデバイスドライバは，メモリ領域の値の変化を監視するブレークポイントをセットしたときのプログラムの実行速度を大幅に向上させます。

デバイスドライバを入れているディレクトリに，TDH386.SYSをコピーします。さらにCONFIG.SYSファイルに，たとえば次のように指定します。

```
DEVICE=¥SYS¥TDH386.SYS
```

このように設定すると，システムの立ち上げ時にTDH386.SYSデバイスドライバがシステムに組み込まれるので，Turbo Debuggerはこのデバイスドライバを使うことができます。

このデバイスドライバインターフェースに関する詳細な説明は，付録Aを参照してください。

注意：［IBMバージョンのみ］ハードウェアデバッグボード（Atron，Periscopeなど）を持っている方は，そのボードをTurbo Debuggerとともに使えるかもしれません。お持ちのボードのTurbo Debuggerとの互換性について，ボードのメーカーに問い合せてください。

次にどこに進めばよいか

Turbo Debuggerを使う準備が整ったので，Turbo Debuggerを使うことができます。このユーザーズガイドは，Turbo言語によるプログラミングの初心者と経験者の両方を対象にしているので，初心者向けの章と経験者向けの章があります。経験に応じて，次のガイドにしたがって先に進んでください。

Turbo言語を学び始めたばかりの人

Turboファミリィの言語のどれかを学び始めたばかりの人は，デバッガの使い方を学ぶ前に，その言語を使って何か小さいプログラムを作ることができるようになりたいと思われでしょう。また自分で作ったプログラムをデバッグするのが，デバッガの使い方を憶えるのに一番いい方法です。ですから，まずその言語を，プログラムが作れるようになるまで習得してください。それから第3章「**チュートリアル**」に進んで，Turbo Debuggerの主な機能をざっと体験してください。それで，あなたの最初のプログラムをデバッグするのに必要な機能について，十分に習得することができます。それからその後の章で，Turbo Debuggerのより高度な機能に進んでください。

Turbo言語をすでに使っている人

Turboファミリの言語に経験のある方は，第2章に進んで，Turbo Debuggerのユーザインターフェースのエキサイティングな新しい機能の説明を読んでください。その後はあなたの好みにより，第3章「**チュートリアル**」に進んでもよいし，すぐに第4章「**Turbo Debuggerの起動**」に進むこともできます。すべてのコマンドを見るには，第12章「**コマンドリファレンス**」を参照してください。

第2章

デバッグとTurbo Debugger

昔，自分は決して間違えないと信じている男がいました。でも彼は間違っていました。それでその男は，デバッガを作りました。それが今，我々がデバッガを持っている理由です。

間違いをしない人間はいません。私たちは皆間違える可能性があります。歩くという簡単なことをしているときでも，つまづくことがあります。

プログラミングの場合には，間違えることが日常茶飯事になります。最初からエラー(誤り）のないプログラムを書くことができるプログラマは，ごく僅かしかいません。間違うことは恥かしいことではありませんし，驚くことでもありません。ただ間違いには，エラーの発見と修正という作業が付随します。そのために，何度も何度も試行錯誤を繰り返さなければなりません。プログラミングの用語では，これをデバッグと言います。

デバッグとは何か？

デバッグとは，プログラムの中のエラー(〝バグ〟と呼ばれる)を見つけ出し，修正することです。プログラムのデバッグのために，最初にそのプログラムを書くのにかかった時間より長い時間を費やすこともまれではありません。デバッグは，精密科学ではありません。多くの場合，一番良いデバッグツールは，あなた自身の頭です。それにもかかわらず，デバッグについてアドバイスできることがいくつかあります(第13章参照)。デバッグの手順は，大まかに次の4つの段階にわけることができます。

1. プログラムにエラーがあることに気がつく。
2. エラーのある場所をつきとめる。
3. エラーの原因を解明する。
4. エラーを修正する。

バグがあるか？

第1の段階は，一見明白です。あるプログラムは，実行するたびにコンピュータが動かなくなります(〝ハングアップ〟する)。また他のプログラムは，クラッシュして画面の全面に無意味な文字を表示します。しかしプログラムにエラーがあることが，すぐにはわからないこともあります。たとえばある数を入力するまで，あるいは出力を注意深く調べるまでは，プログラムは正しく動作しているように見えます。ところがそのような数を入力したときや出力を調べたときに，そしてそのときにのみ，結果が僅かにずれているとか，名簿のミドルネイムの頭文字が間違っているとか，見逃しやすい誤りに気がつきます。

どこにあるか？

第2の段階は，しばしば最も困難です。まず直面することは，非常に小さなプログラム以外は，プログラム全体を頭に入れておくことは不可能だということです。それに対処する最善の方策は，分割して突き止めること，すなわちプログラムをいくつかの部分に分割し，各部分を別々にデバッグすることです。構造化プログラミングは，このデバッグ方法にまさにぴったりです。

原因は何か？

第3の段階，エラー原因の解明は，たぶんデバッグで2番目に難しい部分でしょう。バグがどこにあるかを発見すれば，プログラムが間違った動作をする原因を見つけ出すことは，いつでもいくらか容易です。たとえば，***PrintNames***という名前の手続きにエラーがあることがわかったときは，プログラム全体ではなくその手続きの各行を調べるだけですみます。しかしそうではあっても，エラーの原因は見つけるのが難しく，解明するまでにあれこれ試してみなければならない場合もあります。

バグの修正

最後の段階は，エラーを修正することです。エラーの原因がわかれば，プログラム言語の知識により，虫(バグ)をひねり潰すことができます。エラーを修正したらプログラムを走らせて，次のエラーが現われるかどうか調べます。別のエラーが現われたら，再びデバッグを繰り返します。

この4段階の手順は，プログラムを書いているときにも，何回も繰り返すことになります。たとえば多数の構文エラーのために，それらのエラーを修正するまで，プログラムのコンパイルができないからです。ただしボーランドの言語ソフトウェアには，構文エラーを知らせる構文チェッカが組み込まれて，構文エラーが指摘されたらただちに修正できるようになっています。

しかし他のエラーは，もっと潜み隠れていて発見が困難です。あるものは負の数が入力されるのをじっと待ち受けており，またあるものはなかなか正体を現わさず，プログラマを苦しめます。そういうときにTurbo Debuggerが威力を発揮します。

Turbo Debuggerができること

スタンドアロンデバッガであるTurbo Debuggerを使うことにより，言語コンパイラに組み込まれているデバッガよりずっと強力なデバッグ機能を利用することができます（このような多くの機能を備えたデバッガをコンパイラに付け加えると，プログラムが大きくなりすぎます）。

Turbo Debuggerは，ボーランドまたはマイクロソフトのC，Pascal，アセンブラで作成されたプログラムのデバッグに使用することができます（ただしマイクロソフトの言語で作成されたプログラムは，デバッグする前に，提供された変換ユーティリティを使って変換する必要があります）。他の言語ソフトウェアで作成されたプログラムもデバッグできますが，付録Bで説明するTDMAPユーティリティを使わなければ，アセンブラレベルでのデバッグに制限されるでしょう。

Turbo Debuggerには，プログラムの実行速度を遅くしたり，好きな箇所でプログラムの状態を調べたりすることができる多数の機能があります。また変数に新しい値を設定して，それらの値がプログラムにどのような影響を与えるかを試してみることもできます。これらの機能によりTurbo Debuggerは，デバッグの最も難しい段階，すなわちエラー箇所の発見とエラー原因の追求における強力な武器になります。

これらの機能は，Turbo Debuggerの用語では，**Trace(トレース)**，**Step(ステップ)**，**View(ビュー)**，**Inspect(インスペクト)**，**Change(チェンジ)**，**Watch(ウォッチ)**と呼ばれます。

Trace　プログラムを1行ずつ実行することができます。

Step　手続きや関数の呼び出しをスキップしながら，プログラムを1行ずつ実行することができます。手続きや関数にエラーがないことが確実なときは，それらをスキップすることによりデバッグの要する時間を短縮できます。

View　特別のウィンドウを開き，変数，変数の値，ブレークポイント，スタックの内容，ログ，データ ファイル，ソースファイル，CPUコード，メモリ，レジスタ，数値プロセッサの情報，プログラムの出力などを表示させることができます。

Inspect　プログラムの動作を詳しく調べることができます。また配列や構造体のよ

うな複雑なデータ構造の内容を表示させることができます。

Change 変数の現在の値を，新しい値に変更することができます。

Watch プログラムの変数を指定して，プログラムの実行によるそれらの変数の値の変化を監視することができます。

これらの強力なツールを使ってプログラムをいくつかの部分に分割し，ひとつずつバグがないことを確認していくことができます。このようにして，どんなに大きいプログラムでも，またどんなに複雑なプログラムでも，バグが隠れている場所をつきとめることができます。うっかりミスにより変数に値を再設定する関数がある，無限ループに陥って抜け出せなくなる，不正な再帰ループに引き込まれるなどのいろいろなバグが見つかるかも知れません。どのようなバグであっても，Turbo Debuggerは，バグのある箇所とバグの原因を見つけ出すのを強力に手助けします。

Turbo Debuggerにできないこと

Turbo Debuggerには多数の機能が組み込まれているので，何でもできると思われるかもしれません。でも，Turbo Debuggerにできないことが，少なくとも3つあります。

- Turbo Debuggerには，ソースコードを変更するためのエディタは組み込まれていません。プログラマには一人一人たいてい好みのエディタがあり，それがいちばん使いやすいからです。したがって，Turbo Debuggerにエディタを組み込むことは，メモリの無駄使いになるでしょう。そのかわりFileウィンドウからローカルコマンドを選択することにより，容易にあなたのエディタに制御を渡すことができます（ローカルコマンドについては，すぐ後でもっと詳しく説明します)。Turbo Debuggerは，TDINSTインストールプログラムで指定されたエディタを使用します。
- Turbo Debuggerは，プログラムのソースコードを修正したとき，それをコンパイルすることはできません。ソースコードをコンパイルするには，もとのコンパイラ(Turbo PascalやTurbo C)が必要です。
- Turbo Debuggerを使うことで，考えることが不要になるわけではありません。プログラムをデバッグしているときに最も強力な武器は，考えることです。Turbo Debuggerは強力なツールですが，使い方を誤ると時間と労力の節約はできません。

Turbo Debuggerのユーザインターフェース

Turbo Debuggerは，強力で高度な機能を持っているにもかかわらず，使いやすく，勘だけで使えると言ってもいいほどです。

Turbo Debuggerは，先進的なユーザインターフェース(UI)により，強力な威力と使いやすさを巧みに調和させています。次の節では，Turbo DebuggerのUIの特長について説明します。

Turbo Debuggerの特長

一度Turbo Debuggerを使うと，もう使わずにはいられなくなるでしょう。Turbo Debuggerは，使い易さと便利さを徹底的に追究した設計になっています。この目的を達成するために，次のような強力な機能を備えています。

■便利で合理的なプルダウンメニュー
■Turbo Debuggerのどこにいても，ポップアップメニューが，その時点の状況に応じた適切なコマンドを表示します。コマンドを憶えることも，キーボードから入力することも不要です。
■何かを入力する必要があるときは，同じような状況のもとで入力されたテキストのリストを表示します。この〝ヒストリリスト〟から選択してそのまま入力することもできますし，編集して変更してから入力することもできます。また新しく入力することもできます。
■コマンドやキー入力をスピードアップするためのマクロ定義
■便利で，見やすいウィンドウシステム
■数種類のオンラインヘルプへのアクセス

この章の残りの部分では，これらのTurbo DebuggerのUIの6つの特長について説明します。

メインメニューとプルダウンメニュー

ボーランドの多数の製品と同じように，Turbo Debuggerにも，画面の最上部に沿って表示されたメニューバーからアクセスすることができるメニューシステムを備えています。

メインメニューバーは，どのウィンドウがアクティブであるか(すなわち，どのウィンドウにカーソルがあるか)によらず，いつでも表示されています。メニューバーの各項目は，その項目からアクセスできるプルダウンメニューを示します。

どれかのウィンドウからメインメニューバーのプルダウンメニューに入るには，次の3つの方法があります。

■***F10***を押し，希望するメニューにカーソルを移動して**リターン**を押します。

■***F10***を押し，希望するメニューの名前の先頭の文字(***F***, ***V***, ***R***, ***B***, ***D***, ***W***, ***O***)を押します　。

■***GRPH***を押しながらメインメニューコマンドの先頭の文字(***F***, ***V***, ***R***, ***B***, ***D***, ***W***, ***O***)を押します。するとその文字により指定されたプルダウンメニューがアクティブになります。たとえば**Turbo Debugger**システムのどこにいても，***GRPH-F***を押すとFileメニューに入ります。

コマンドを選択しないでメニューバーから出るには***ESC***を押します。

メニューの中にいるときは，

■***ESC***と押すとメニューから抜け出します。第2レベルのメニューの中にいるとき以外は，その前にアクティブであったウィンドウに戻ります。

■どのメニューレベルにいるときでも，***F10***を押すとその前にアクティブであったウィンドウに戻ります。

■**左右の矢印キー**を使うと，プルダウンメニューから隣のプルダウンメニューに移ることができます。

■***HOME***を押すとメニューの最初の項目に移ります。***SHIFT-HOME***(IBM PCでは***End***)を押すとメニューの最後の項目に移ります。

メインメニューの中のいくつかのコマンドには，ショートカットキーコマンド（**ホットキー**とも呼ぶ）が割り当てられています。ホットキーを表示するのが適当なときには，メニューコマンドの右側にホットキーが表示されます。

第12章の図12.1，12.2，12.3には，Turbo Debuggerのプルダウンメニューの木構造の全体が示されています。また表12.1には，すべてのホットキーが掲載されています。さらにTurbo Debuggerで使用できるすべてのコマンドの要約説明も，第12章にあります。

状況感知機能

ボーランドの便利なプルダウンメニューシステムに加えて，メニューの数を減らすことにより混乱を少なくし使い方を憶えやすくする，もう1つの強力な機能があります。

この機能を理解するためには，Turbo Debuggerが状況を感知する能力を備えた(context-sensitive)プログラムであることを知る必要があります。Turbo Debuggerは，どのウィンドウが開かれたか，どのテキストが選択されたか，ウィンドウのどの部分（すなわちどの

ペイン)にカーソルがあるかを監視しています。言い換えればTurbo Debuggerは，ユーザがコマンドを選択するときに，ユーザが見ているものと，カーソルの置かれている場所を正確に認識しています。そしてコマンドに応答するときにこの情報を利用します。

たとえば，あるPascalのプログラムに，次のような行があるとしましょう。

```
MyCounter[TheGrade] := MyCounter[TheGrade] + 1;
```

Turbo Debuggerを使うとわかりますが，データ構造に関する情報を得ることは簡単です。ただ***CTRL-I***を押す(Inspectコマンドを選択する)だけです。カーソルが*MyCounter*の上にあるときは，Turbo Debuggerは，この配列変数の内容をすべて表示します。この配列名と配列の添字を選択して（ハイライト表示される）***CTRL-I***を押したときは，デバッガは，あなたが1つのメンバーだけを調べたいと思っていることを察知して，その配列のその要素だけを表示します。また配列を調べているときに***CTRL-I***を押すと，1つのメンバーを見ることができます。このようにして，だんだんとプログラムの細かい部分を調べることができます。

この状況感知能力が，Turbo Debuggerを非常に使いやすくしています。またメニューコマンドの名前やわかりにくいコマンドラインスイッチを，記憶したり入力したりするわずらわしさから解放してくれます。ただ調べたいと思う変数や関数にカーソルを移動して(または***INS***キーを使って選択して)，コマンドを呼び出す(たとえばInspectコマンドに対しては***CTRL-I***を押す）だけで済みます。

この状況感知能力により，ユーザは楽になりますが，コマンドの説明書を書く作業は困難になります。なぜならコマンド，たとえば***CTRL-I***の結果はいつでも同じではなく，カーソルの位置や選択されているテキストによって変わるからです。

ローカルメニュー

Turbo Debuggerの状況感知能力は，ポップアップメニューの表示にも利用されています。

Turbo Debuggerのポップアップメニューは，表示内容がカーソルの位置により異なることを示すために，〝ローカル〟メニューと呼びます。ポップアップ(ローカル)メニューとプルダウン(グローバル)メニューを混同しないようにしてください。この2つのメニューを比較すると，次のようになります。

プルダウン(グローバル)メニュー

■プルダウンメニューは，***F10***を押し，矢印キーを使うかまたはメニュー名の先頭文字を

入力することによりアクセスするメニューです。

■プルダウンメニューの名前は，いつも画面の最上部のメニューバーの中に表示されています。

■プルダウンメニューの内容は変化しません。

■いくつかのメニューコマンドには, Turbo Debuggerのどこにいるときでも有効なホットキーがあります。

ポップアップ(ローカル)メニュー

■ポップアップメニューは，***GRPH-F10***または***CTRL-F10***を押すことにより呼び出すことができます。

■メニューの表示位置や内容が，選択されているテキストやカーソルの位置により変化します。

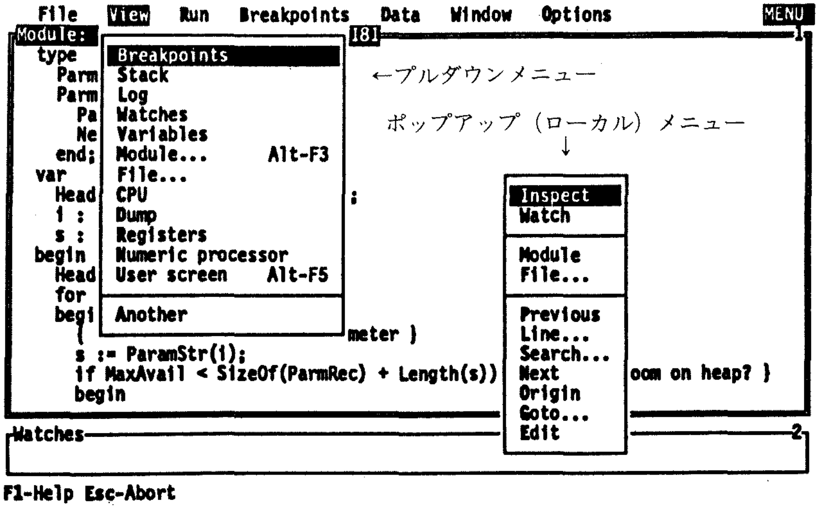
■ポップアップメニューの内容は，いつも同じとは限りません（ただし，ほとんどすべてのポップアップメニューに現われる核となるコマンドがあります)。同じ名前のコマンドでも，状況によって結果が異なる場合があります。

■ポップアップメニューのすべてのコマンドには, ***CTRL***とコマンドの先頭文字との組み合わせからなるホットキーがあります。

ホットキー，たとえば***CTRL-S***は，ある状況における意味と，別の状況における意味がまったく異なります。ただしそれでも先に説明したように，核となるコマンドには，ポップアップメニューによらず，意味に一貫性があります。たとえばGotoコマンドやSearchコマンドは，異なるペインから呼び出されたときにも同じことをします。

これらの2種類のメニューを示す合成画面を次に示します（実際には2種類のメニューが同時に画面に表示されることはありません)。

図2.1　プルダウンメニューとポップアップメニュー



ユーザの立場からは，ローカルメニューは大変便利です。その時点で適切なすべての選択可能なコマンドが，一目でわかるように表示されます。このようなローカルメニューにより，不適切なコマンドの選択が防止され，また各メニューが小さく見やすくなります。

ヒストリリスト

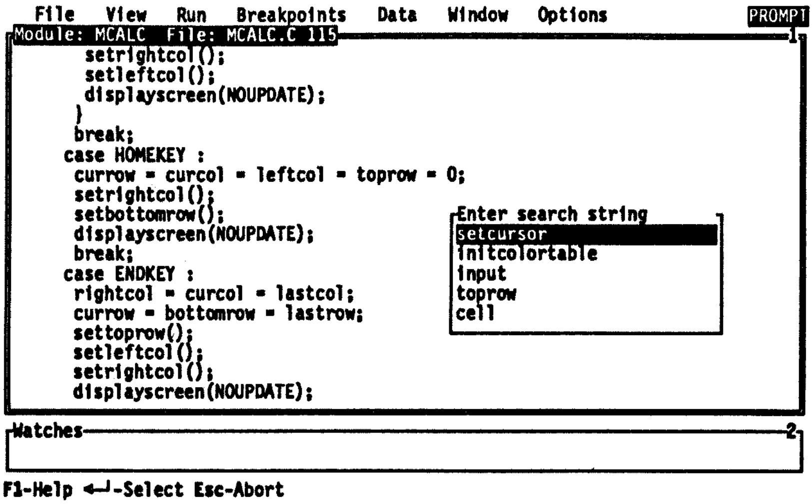
Turbo Debuggerのユーザインターフェースには，メニューシステムと状況感知能力の他にもう1つの特長があります。それは〝ヒストリリスト〟です。

Turbo Debuggerは，「ユーザのキーボードからの入力はできるだけ少なくしなければならない」という哲学にしたがって，プロンプトボックスに入力されたものを記憶し，そのボックスが再び呼び出されたときはいつでも，記憶したテキストを表示します。

たとえば，*ReturnOnInvestment*という名前の関数を探すときは，最初はその名前の全体または一部を入力しなければなりません。しかし次にそのプロンプトボックスを見るとき，たとえば*myPercentage*という変数を探すためにそのプロンプトボックスを開いたときは，そのボックスの中に*ReturnOnInvestment*という文字列があるのに気づくでしょう。さらに別の文字列を探すときは，先に入力した2つの文字列がそのボックスに現われます。このリストは，Searchコマンドを使うたびに，だんだん大きくなります。

次の図は，サーチプロンプトボックスとその中に表示されたヒストリを示します。

図2.2　プロンプトボックスとヒストリリスト



矢印キーを使ってヒストリリストの中のテキストを選択して**リターン**キーを押すと，そのテキストの検索を開始することができます。ヒストリリストは，このようにキーボードからのテキストの入力に対するショートカット（近道）として使用することができます。ヒストリリストの項目をそのまま使ったときは，その項目がリストの一番上に移動します。テキストを編集することもできます。たとえば*myPercentage*の*my*を*his*に変えるだけで，*hisPercentage*に変更することができます。またある項目がハイライト表示されているときに新しい項目の入力を始めると，ハイライト表示されている項目の上に上書きすることができます。Moduleウィンドウの中のカーソルが置かれている文字列は，サーチリストの先頭に表示されます。

Turbo Debuggerは，それ以前に入力したテキストのうち新しいものを10個表示します（サーチリストの大きさはTDINSTプログラムを使って変更することができます）。

Turbo Debuggerは，ほとんどのプロンプトボックスのために，別々のヒストリリストを作成します。そのため，テキストをサーチするために入力したテキストが，たとえば，あるラベル（プログラム行）に移動するためのプロンプトボックスを乱だすことはありません。

マクロの作成

マクロは，簡単に言えば，あなたが定義するキーストロークのショートカットです。一連のTurbo Debuggerのコマンドや文字列をただ1つのキーに定義し，そのキーの一押しで定義したコマンド列や文字列を入力することができます。

マクロを作成するには，次のようにします。まず**F10**を押してメニューバーをアクティブにし，**O**を押してOptionsメニューを選択し，Optionsメニューから**Macros**を選択します。すると4つのコマンド**Create**，**Remove**，**Delete all**，**Stop recording**が表示されます。ここでCreateを選択します。するとTurbo Debuggerは，その後の操作を保存したいキーの入力を求めてきます。それに答えて，あまり使用しないキーまたは思い出しやすいキー(たとえば，プログラムを実行させるためのコマンド列に対して**SHIFT-F1**)を押します。次にそのキーに保存したいすべての操作を行ないます。記録セッションを終了するには，新しく定義されたマクロキー(**SHIFT-F1**)をもう一度押します。

一連のステップを繰り返していることに気がついたときはいつでも，「これにマクロを使うことができないだろうか」と自問してください。たとえば，プログラムにウィンドウの右端まで達するコメント行がたくさんあるかどうかにより，ウィンドウの大きさを変更したり，ウィンドウを動かしたりしますか。そうするなら，Moduleウィンドウの幅を半分にするマクロ(このマクロは**SHIFT-F2**として定義します)と，幅をいっぱいにするマクロ(このマクロは**SHIFT-F3**として定義します)を作成します。このようにすると，ウィンドウの幅を簡単に切り替えることができます。

ウィンドウ

今では，多数のプログラムがウィンドウを使っています。しかしTurbo Debuggerは，ウィンドウをより巧みに使っています。Turbo Debuggerは，メニュー，プロンプトボックス，およびウィンドウに，すべての情報やデータを表示します。ウィンドウは，内部に表示される情報の種類によって分類することができます。すべてのウィンドウは，メニューコマンド(またはホットキーによるショートカット)により開閉することができます。ほとんどのウィンドウは，プルダウンViewメニューから開くことができます(Viewメニューからは12種類のウィンドウを開くことができます)。その他にInspectorウィンドウと呼ばれるもう一種類のウィンドウがあります。この種類のウィンドウは，**Data/Inspect**を選択するか，またはポップアップメニューからInspectを選択することにより開くことができます。

Viewメニューから開くことができるウィンドウ

次に，Viewメニューからコマンドを選択することにより開くことができる，12種類のウィンドウについて説明します。これらのウィンドウは，***F3***を押すかWindow/Closeを選択することにより，閉じることができます。もし誤ってウィンドウを閉じたときは，Window/Undo closeを選択することにより，そのウィンドウを再び開くことができます(ウィンドウの内容は，ウィンドウを閉じたときとまったく同じです)。ただしこの方法で元に戻せるのは，最後に閉じたウィンドウだけです。

Moduleウィンドウ

デバッグしているプログラムを表示します。表示されているモジュールの中を動き回り，変数の値を調べたり，Watchesウィンドウに追加したりすることができます。

他のどのウィンドウよりもModuleウィンドウの中で，多くの時間を費やすことになるでしょう。ですから十分な時間をかけて，このウィンドウのすべてのローカルメニューコマンドを完全に習得してください。

このウィンドウは，***GRPH-F3***を押して開くこともできます。

Moduleウィンドウとそのローカルコマンドは，第8章に詳しく説明されています。

Watchesウィンドウ

変数とそれらの現在の値を表示します。希望する変数の上にカーソルを置いてウォッチコマンド(***CTRL-W***)を選択することにより，その変数をこのウィンドウに入れることができます。

Breakpointsウィンドウ

現在設定されているブレークポイントを表示します。Turbo Debuggerのブレークポイントは，普通，ブレークポイント，ウォッチポイント，トレースポイントと区別して呼ばれている機能をすべて含みます。

このウィンドウを使って，ブレークポイントを変更したり，削除したり，追加したりすることができます。

Breakpointsウィンドウとブレークポイントの働きについての詳細な説明は，第7章を参照してください。

Stackウィンドウ

スタックの現在の状態を表示します。このウィンドウの中の関

数や手続きをカーソルで指定して，その関数や手続きに関するより詳しい情報を得たり，ローカル変数や呼び出しの引数の型を調べたりすることができます。
Stackウィンドウの詳細な説明は，第5章を参照してください。

Logウィンドウ

ログの内容を表示します。ログには，プログラムが停止した原因やブレークポイントの結果，記録を取ることにした変数の値などが記録されます。ログは，あらかじめ決められている大きさを越えると，新しい内容が書き込まれるにつれて古い内容が失われます。
このウィンドウにより，プログラムの動作を過去にさかのぼって追跡することが可能になります。
Logウィンドウの詳細な説明は第7章を参照してください。

Variablesウィンドウ

ユーザプログラムの中の現在の場所からアクセス可能なすべての変数を表示します。左側のペインにはグローバル変数が，右側のペインにはローカル変数が表示されます(もしあれば)。
このウィンドウは，名前が正確に思い出せない関数や変数を見つけ出したいときに便利です。そのようなときは，このウィンドウを開いて，探しているものをすばやく見つ出すことができます。
Variablesウィンドウの詳細な説明は第5章を参照してください。

Fileウィンドウ

ディスクファイルの内容を表示します。ファイルの内容は16進数バイト形式またはASCIIテキスト形式で見ることができます。特定の文字の並びまたはバイトの並びを検索したり，ディスクファイルの一部を直接パッチすることもできます。Fileウィンドウを開くには，***F2***を押します。
このウィンドウは，ディスクファイルを扱うプログラムをデバッグしていて，ファイルの内容を変更することによりプログラムの動作を変えたいときに便利です。また，ファイルの内容の間違いを訂正したり，プログラムにより作成されたファイルの内容が正しいことを確認するために内容を調べたりすることもできます。
Fileウィンドウの詳細な説明は，第8章を参照してください。

CPUウィンドウ

CPUの現在の状態を表示します。このウィンドウには，5つのペインがあります。機械語が逆アセンブルして表示されるペイン，データが16進数バイト形式で表示されるペイン，スタックの内容が16進数バイト形式で表示されるペイン，CPUのレジスタの内容が表示されるペイン，およびCPUのフラグの状態が表示されるペインです。

CPUウィンドウは，ソースコードの1行に対応する一群の命令や，構造体を構成するバイトをウオッチしたいときに便利です。またアセンブリコードがわかる方には，発見が困難ないくつかのタイプのバグを突き止めるときに，助けになります。ほとんどのプログラムのデバッグでは，このウィンドウを使う必要はないでしょう。

CPUウィンドウとアセンブラレベルのデバッグは，第10章に説明されています。

Dumpウィンドウ

メモリの指定された領域の内容を表示します。データはASCII形式，16進数バイト形式，ワード形式，ダブルワード形式，それにすべての浮動小数点形式で表示させることができます（このウィンドウは，CPUウィンドウのDataペインと同じものです）。メモリのある領域の内容だけを見たいときは，CPUウィンドウのかわりにこのウィンドウを使うことができます。このウィンドウのローカルメニューには，表示されたデータを変更したり，データの表示形式を切り替えたり，データのブロックを操作したりするためのコマンドがあります。

このウィンドウについての詳しい説明は第10章を参照してください。

Registersウィンドウ

CPUのレジスタとフラグの内容を表示します。このウィンドウには，2つのペインがあります。各ペインはそれぞれCPUウィンドウのRegistersペインおよびFlagsペインと同じです。CPUのレジスタやフラグの内容だけを見たいときは，CPUウィンドウのかわりに，このウィンドウを使うことができます。このウィンドウのローカルメニューコマンドにより，レジスタやフラグの内容を変更することができます。

このウィンドウについての詳しい説明は第10章を参照してください。

Numeric Processor

数値演算コプロセッサの状態を表示します。このウィンドウにウィンドウプロセッサは3つのペインがあります。浮動小数点レジスタの内容を表示するペイン，ステータスフラグの値を表示するペイン，コントロールフラグの値を表示するペインです。このウィンドウは，浮動小数点数を扱うプログラムの診断を支援します。このウィンドウを本当に有効に利用するためには，数値演算コプロセッサの内部動作についてかなりの理解を必要とします。

Numeric Processorウィンドウの使い方の詳しい説明は第11章を参照してください。

User screenウィンドウ

デバッグしているプログラムの出力画面を表示します。このウィンドウに表示される画面は，プログラムがTurbo DebuggerではなくDOSのもとで動作しているときの画面と正確に同じです。

このウィンドウを使うと，プログラムにより画面に表示したいものが正しく表示されているかどうか調べることができます。このウィンドウは，***GRPH-F5***により開くこともできます。User Screenウィンドウを見た後は，任意のキーを押すとデバッガの画面に戻ります。

CPU，File，Moduleの3種類のウィンドウは，View/Anotherを選択することにより複数個開くことができます。この機能により，アセンブリコードの異なる部分，プログラムが使用または作成する複数の異なるファイル，複数のプログラムモジュールなどを同時に見ることができます。

Turbo Debugger自身がこれらのウィンドウの1つを開いても，驚かないでください。Turbo Debuggerは，コマンドに応答してこれらのウィンドウを開くことがあります（たとえば，Codeペインの中にいてソースファイルを見るための選択をすると，Turbo Debuggerは自動的にCPUウィンドウまたはModuleウィンドウを開きます）。

Inspectorウィンドウ

Inspectorウィンドウは，指定された変数の現在の値を表示します。このウィンドウは，けっして複数のペインに分割されません。Inspectorウィンドウを閉じるには，普通は***ESC***を押します。複雑なデータ構造を調べるために2つまたはそれ以上のInspectorウィンドウを連続して開いたときは，***F3***を押すか，Window/Closeコマンドを選択することにより，すべてのInspectorウィンドウを一度に閉じることができます。

変数や式の値，配列の要素などを見たいときは，Inspectorウィンドウを開いて表示させることができます。

Inspectorウィンドウは，単純なスカラ(**int, float,** その他)はもちろん，ポインタ，配列，レコード(Pascalのデータ型の1つ)，構造体，共用体などの複雑なデータ構造を表示することができます。各データ型は，プログラムのソースコードで見なれている書式に近似した書式で表示されます。

Inspectorウィンドウは，Viewメニューから開くウィンドウと異なり，Inspectコマンドを選択するだけで，いくつでも開くことができます(Viewメニューから開かれるウィンドウのうち，モジュール，ファイル，CPUの3種類のウィンドウだけは，View/Anotherコマンドを選択することにより複数個開くことができます)。

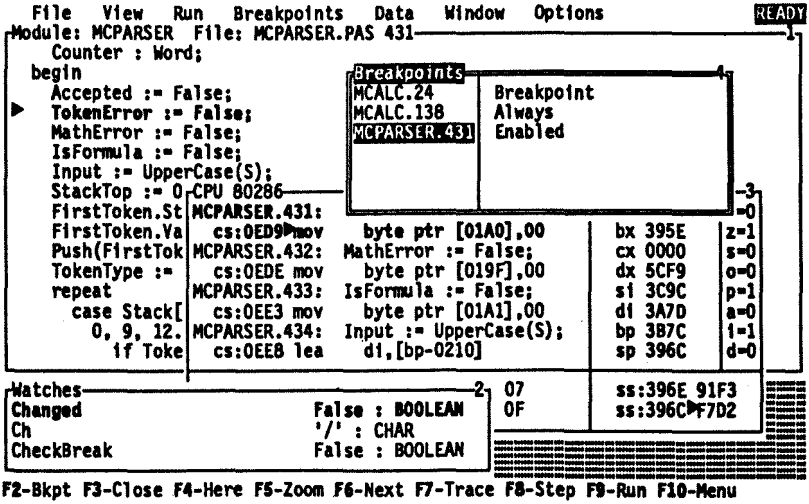
アクティブウィンドウ

Turbo Debuggerでは，複数のウィンドウを同時に開くことができますが，アクティブなウィンドウは1つだけです。アクティブなウィンドウは，次の基準により見つけ出すことができます。

- アクティブウィンドウは，太い線(IBM PCでは二重線)で囲まれている。
- アクティブウィンドウは，タイトルがハイライト表示されている。
- アクティブウィンドウは，その中にカーソルやハイライトバーがある。
- ウィンドウが重なりあっているときは，一番上のウィンドウである。

コマンドを選択(入力)したり，テキストを入力したり，表示をスクロールしたりするとき，影響があるのはアクティブウィンドウだけです。他のウィンドウには影響しません。

図2.3　アクティブウィンドウ



ウィンドウ間の移動

GRPH（***Alt***）と，各ウィンドウの右上隅に表示されているウィンドウ番号を指定する数字キーを組み合わせて押すと，ウィンドウ番号1から9のウィンドウのうち，任意のウィンドウをアクティブにすることができます（ウィンドウは開かれた順に番号が付けられます。普通はModuleウィンドウがウィンドウ1に，Watchesウィンドウがウィンドウ2になります。その後で開くウィンドウは，3およびそれ以上の番号になります)。

GRPH-0を押すと，現在開かれているウィンドウのリストが表示されます。このリストからウィンドウを選択して**リターン**を押すと，そのウィンドウがアクティブになります。

また***F6***を押すと，開いているウィンドウを順番にアクティブにすることができます。この機能は，開こうとするウィンドウの番号が隠れているときに便利です。

ウィンドウに複数の**ペイン**(たとえばウィンドウが格子入りの窓ならばペインは1枚の窓ガラスに相当します。異なる種類のデータに割り当られたウィンドウの領域)があるときは，***TAB***または***SHIFT-TAB***を押すか，Window/Next pane cycleを選択することにより，他のペインに移ることができます。

ペインからペインに移るときに，あるペインではブリンク(点滅)するカーソルが現われ，きは，キーパッドのキーを使ってテキストを動き回ることができます（たとえば***ROLLUP***,

CTRL-HOME，***CTRL-ROLLDOWN***は，それぞれ，カーソルを1画面分下方，ペインの最上部，リストの先頭行に移動します)。またWordStarと同じようなキーの組み合わせにより，ペインの中を動き回ることができます。ペインの中を動き回るために使用できるキーストロークコマンドについては，第12章の表を参照してください。

カーソルのかわりにハイライトバーが現われたときは，カーソルが現われたときと同じように矢印キーを使って動き回ることができますが，さらにいくつかの特別のキーを使用することができます。

たとえば各項が英字の文字列からなるリストでは，〝キー入力による選択〞ができます。英字を入力するたびに，ハイライトバーが，それまでに入力した文字で始る最初の項目に移動します。ハイライト表示された項目上のカーソルの位置は，その名前のうち何文字を入力したかを示します。ハイライトバーが希望する項目に移動したら，検索が完了します。この〝キー入力による選択〞機能により，リストから1つの項目を選択するためのキーストロークは，大幅に少なくなります。

GRPH-F10または***CTRL-F10***を押すと，ローカルメニューが表示されます。

いくつかのウィンドウやペインでは，**リターン**キーが，ハイライト表示されている項目を調べたり変更したりするコマンドへのショートカットになっています。また***DEL***キーは，しばしばハイライト表示されている項目を削除するコマンドへのショートカットになっています。さらに数値や文字列などを入力することが，検索コマンドや変更コマンドへのショートカットになっているウィンドウやペインもあります。

このようなショートカットと，それにより起動されるローカルメニューコマンドについては，第12章のローカルメニューコマンドの要約を参照してください。

ウィンドウの再設定と保存

Turbo Debuggerが新しいウィンドウを開くとき，そのウィンドウはカーソルの現在位置の近くに現われます。ウィンドウの大きさは，指定されたウィンドウの種類に適したデフォルト（省略時）の大きさです。ウィンドウの位置と大きさはWindow/Move/Resizeコマンドにより調節することができます。このコマンドを選択すると，アクティブウィンドウの枠が1本線に変ります。次に矢印キーを使って，画面上でそのウィンドウの位置と大きさを変更します。満足できる状態になったら**リターン**を押します。***GRPH-XFER***(IBM PCでは***Scroll Lock***)は，Move/Resizeコマンドのショートカットです。

ウィンドウを素早く拡大または縮小したいときは，***F5***を押すことにより〝ズーム〞または〝アンズーム〞することができます。

ウィンドウを気に入るように配置したら，Options/Save optionsコマンドを使って，その画面の設定を保存することができます。画面の設定をTDCONFIG.TDに保存したとき

は，DOSからTurbo Debuggerを起動するといつでもその画面が現われます。Turbo Debuggerが起動時に自動的にロードするのはTDCONFIG.TDだけです。他のコンフィギュレーションファイル(TDCONFIG.TDのコピーに別の名前を付けたファイル)に保存した異なる設定は，メインメニューからOptions/Restore optionsを選択するか，Turbo Debuggerを起動するときに-cコマンドラインオプションで指定することにより，読み出すことができます。

ヘルプの呼び出し

すでに見ておわかりのように，Turbo Debuggerは，デバッグを容易にするために，いろいろとユーザの負担を肩がわりします。コマンドの記憶は最小限で済みます。以前に入力したことがあるものは，リストから選択するだけで入力できます。マクロを定義することができます。信じられないほど自由なウィンドウ制御ができます。しかしTurbo Debuggerは多数の機能とコマンドを持つ複雑なプログラムですから，それでもなおユーザが陥るかもしれない混乱を防ぐために，次のようなヘルプ手段によって導いてくれます。

■画面の右上隅にハイライト表示されるアクティビティインジケータに，いつも現在の状態が表示されます。たとえば，カーソルがウィンドウの中にあるときは，アクティビティインジケータにREADYと表示されます。メニューが表示されているときは，MENUと表示されます。プロンプトボックスの中にいるときは，PROMPTと表示されます)その他に表示される可能性があるのは，MOVE/RESIZE, MOVE, およびERRORです)。Turbo Debuggerの中で現在の状態がわからなくなったときは，まずアクティビティインジケータを見てください。

■アクティブウィンドウは，いつでも，重なり合っているウィンドウの一番上のウィンドウであること，枠が2重線であることを思い出してください。

■***F1***を押すことにより，詳細なヘルプシステムにアクセスすることができます。

■画面の最下行には，いつでも，その時点で有効なキーストロークコマンドのクイックリファレンスが表示されます。状況が変化したとき，および***GRPH***または***CTRL***を押したときは，この行の表示が変化します。

最後の2つのヘルプ機能については，以下の説明を読んでください。

オンラインヘルプ

キーを1回押すだけで，オンスクリーンヘルプが表示されます。エラーメッセージやプロンプトが表示されているときだけでなく，メニューやウィンドウの中にいるときにもヘル

プを利用することができます。

F1を押すと，ヘルプウィンドウが開いて現在の状況にぴったりの情報を表示します。ヘルプ画面の中には，ハイライト表示されたキーワードを含むものがあります。これらの項目に関しては，さらに詳しいヘルプを見ることができます。矢印キーを使ってそれらのキーワードのどれかに移動し**リターン**を押すと，その項目のヘルプ画面が表示されます。***HOME***キーおよび***SHIFT-HOME(End)***キーを押すと，それぞれ，現在の画面の最初のキーワードおよび最後のキーワードに移動することができます。

直前のヘルプ画面に戻るには，***GRPH-F1***を押します。ヘルプシステムの中にいるときは，***ROLLDOWN(PgUp)***を押すとそれまでの最大20画面まで順にさかのぼることができます。***ROLLUP(PgDn)***は，状況に特有の1組のヘルプ画面の中にいるときにのみ有効です。ヘルプインデックスにアクセスするには，ヘルプシステムの中で***F1***を押します。ヘルプから抜け出すには***ESC***を押します。

最下行

Turbo Debuggerの中にいるときはいつでも，画面の最下行にクイックリファレンスが表示されます。このクイックリファレンス行には，その時点で有効なキーストロークコマンドが表示されます。

通常は，その時点でファンクションキーにより入力することができるコマンドが表示されます。

図2.4　通常のクイックリファレンス行

```
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Run F9-Menu
```

GRPHキーを1, 2秒間押し続けると，***GRPH***キーとファンクションキーの組み合わせにより入力されるコマンドが表示されます。

図2.5　GRPHが押されているときのクイックリファレンス行

```
Alt: F2-Bkpt at F3-Mod F4-Anim F5-User F6-Undo F7-Instr F8-Rtn F9-To
```

CTRLキーを1，2秒間押し続けると，***CTRL***キーと文字キーの組み合わせにより入力されるコマンドが表示されます。表示されるコマンドは，その時点でローカルメニューにより選択できるコマンドです。したがって，現在のウィンドウおよび現在のペインに応じて変化します。コマンドの数が多くてクイックリファレンス行に表示しきれないときは，先頭の文字だけが表示されます。***GRPH-F10***または***CTRL-F10***を押してローカルメニュー全体を

ポップアップさせることにより，そのローカルメニューから選択できるすべてのコマンドを見ることができます。

図2.6　*CTRL*が押されているときのクイックリファレンス行

```
Ctrl: I-Inspect W-Watch M-Module F-File P-Previous L-Line S-Search N-Next
```

第3章

チュートリアル

この章では，最初のプログラムをデバッグするのに十分な知識が身につけられるように構成されています。もしあなたがTurbo Debuggerを使ってみたくてしかたがなくて，そしてマニュアルから先に読むタイプでないなら，まずこの章でTurbo Debuggerの使い方を習得してください。この章に説明されている基本的な知識をいったん習得すれば，巧みに構成されたユーザインターフェースと状況を感知するヘルプシステムの助けにより，使いながら学んでいくことができます。

この章では，Turbo Debuggerのすべての基本的な機能を実際に使ってみることができます。マスターディスクに入っている例題プログラム（Cのプログラムが1つとPascalのプログラムが1つ）について説明したあと，次のことを説明します。

- ■プログラムを実行および停止する方法
- ■変数の値を調べる方法
- ■配列や構造体などの複雑なデータ構造を見る方法
- ■変数の値を変更する方法

例題プログラム

例題プログラム（C用はTCDEMO.C，Pascal用はTPDEMO.PAS）を使って，プログラムをデバッグするために知っていなければならない2つの最も重要なことを説明します。プログラムを実行および停止する方法と，プログラムの変数の値やデータ構造を調べる方法です。例題プログラムそれ自体は，実用を目的としたものではありません。プログラムのコードやデータ構造には，ただTurbo Debuggerの機能を説明するためだけに入れてあるものがあります（たとえば，例題プログラムを使ってコマンドラインパラメータにアクセスする方法が説明されていますが，それらのパラメータは実際には使用されません）。

各例題プログラムは，テキストを何行か入力することを求めて，入力された語数と文字数をカウントします。さらにプログラムの最後の部分で，そのテキストに関する集計（1行当たりの平均文字数と各文字の出現頻度）を表示します。

カレントディレクトリに，このチュートリアルのために必要な2つのファイルが入っていることを確認してください。Cの例題を使用するときは，TCDEMO.CとTCDEMO.EXEが必要です。またPascalの例題を使用するときは，TPDEMO.PASとTPDEMO.EXEが必要です。

Cの例題プログラムを使うときは，次のように入力して**リターン**を押してください。

```
TD TCDEMO
```

Pascalの例題プログラムを使うときは，次のように入力して**リターン**を押してください。

```
TD TPDEMO
```

Turbo Debuggerは，例題プログラムをロードし，プログラムの先頭にカーソルを置きます。

図3.1　TCDEMOを表示する起動時の画面

```
  File  View  Run  Breakpoints  Data  Window  Options                    READY
Module: TCDEMO  File: TCDEMO.C 31                                            1
  static void showargs(int argc, char *argv[]);

  /* program entry point
   */
▶ int main(int argc, char **argv) {
        unsigned int  nlines, nwords, wordcount;
        unsigned long totalcharacters;

        nlines = 0;
        nwords = 0;
        totalcharacters = 0;
        showargs(argc, argv);
        while (readaline() != 0) {
                wordcount = makeintowords(buffer);
                nwords += wordcount;
                totalcharacters += analyzewords(buffer);
                nlines++;
        }
Watches                                                                      2

F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

この画面は，メインメニュー行，Moduleウィンドウ，Watchesウィンドウ，ヘルプ行（クイックリファレンス行）から構成されています。

チュートリアルから出てDOSに戻るには，いつでも***GRPH-X***を押します。チュートリアルにしたがって進んでいるうちに，混乱してどうしてよいかわからなくなったときは，***CTRL-F2***を押してください。***CTRL-F2***を押すと，プログラムが再ロードされるので，もう一度最初からやり直すことができます。ただし***CTRL-F2***はブレークポイントやウォッチをクリアしません。プログラムを再ロードしブレークポイントやウォッチをクリアするには，***GRPH-FL***を使わなければなりません。

カレントウィンドウ，コマンド，プロンプト，エラーメッセージなどについてヘルプが必要なときは，***F1***を押してください。メニューシステムの中を動き回り，各コマンドのところで***F1***を押してそのコマンドの機能の要約を見ることにより，多くのことを学ぶことができます。

Turbo Debuggerを使う

画面の最上行はメニューバーです。メニューバーに表示されているプルダウンメニューを開くには，***F10***を押してメニューバーに移ってから選択するか，***GRPH***キーを押しながらメニュー名の先頭文字を押します。

ここで***F10***を押してみてください。Moduleウィンドウからカーソルが消え，メインメニューのFileコマンドがハイライト表示されることに注目してください。また画面の最下行の表示も変化して，Fileメニューのコマンドを選択するために使用できるキーが表示されます。表示されるキーは***F1***と***ESC***だけです。

左右の矢印キーを使ってメインメニュー上を動き回ってみてください。下向きの矢印キーを押して，ハイライト表示されているプルダウンメニューを開いてください。

ESCを押して，メニューシステムのレベルをさかのぼってください。プルダウンメニューが開いた状態で***ESC***を押すと，プルダウンメニューが閉じてメインメニューに戻ります。メインメニューのハイライト表示されている項目が1つだけのときは，***ESC***を押すと，Moduleウィンドウに戻ります。***ESC***を押して，Moduleウィンドウに戻ってください。

ヘルプ行

画面の最下行は，その時点のファンクションキーの働きを示します。この行の表示は，何を入力しているか（メニューコマンド，プロンプトへの応答，その他）によっても変わります。

GRPHを1,2秒間押し続けてください。最下行の表示が変化して，***GRPH***と一緒に押したときの各ファンクションキーの働きが表示されることに注目してください。

今度は***CTRL***を1秒間押し続けてください。最下行の表示が変化して，現在のウィンドウまたは**ペイン**で使用できる**ローカルメニューコマンド**へのショートカットが表示されます。このショートカットの表示は，ウィンドウの種類とペインにより変化します。これについては，あとでさらに詳しく説明します。

ウィンドウ

ウィンドウ領域は，画面の大部分を占めます。この領域が，ウィンドウを通してあなた

のプログラムを調べる場所です。

画面の表示は，2つのウィンドウ（1つのModuleウィンドウとWatchesウィンドウ）で始ります。あなたが新しいウィンドウを開くか，これらの2つのウィンドウの位置や大きさを調節するまで，これらのウィンドウは〝タイリング（並置）〟のままです。タイリングとは，ウィンドウが重なり合うことなく画面全体を覆っていることを言います。したがって新しいウィンドウは，自動的に，すでに存在するウィンドウに重なります。

Moduleウィンドウは，枠が太線（IBM PCでは2本線）で，タイトルがハイライト表示されていることに注目してください。このことは，現在Moduleウィンドウがアクティブウィンドウであることを示します。

カーソルキー（矢印，***HOME***，***ROLLUP***，***ROLLDOWN***など）を使うとアクティブウィンドウの中を動き回ることができます。Moduleウィンドウの中を動き回ってみてください。

F6を押すと，別のウィンドウに切り替わります。押してみてください。ウォッチウィンドウがアクティブになります。Watchesウィンドウの枠が太線（または2本線）に，タイトルがハイライト表示に変わることに注目してください。

Viewメニューコマンドを使うと，新しいウィンドウを開くことができます。***GRPH-V***を押してViewメニューを開き，***S***を押しみてください。Viewメニューが消えると，Moduleウィンドウの上にStackウィンドウが現われます。

アクティブウィンドウを閉じるには***F3***を押します。押してみてください。Stackウィンドウが消えるでしょう。

Turbo Debuggerは，最後に閉じられたウィンドウを保存します。これによって必要があるときは，元に戻すことができます。閉じたウィンドウをもう一度開くには，***GRPH-W***を押してWindowメニューを開き，***U***を押してUndo closeコマンドを選択します。やってみてください。Stackウィンドウが再び現われたでしょう。最後に閉じたウィンドウをもう一度開くために，***GRPH-F6***を押すこともできます。

Windowメニューには，すでに開かれているウィンドウを動かしたり，その形状を変更することができるコマンドが入っています。ウィンドウの調節には，***GRPH-XFER***（IBM PCでは***ScrollLock***）を使うこともできます。

GRPH-W Mを押してください（WindowメニューのMove/Resizeコマンドが選択されます）。次に矢印キーを使って，アクティブウィンドウ（Stackウィンドウ）の位置と大きさを変更してください。***SHIFT***を押しながら矢印キーを使うと，ウィンドウの大きさを変えることができます。新しい大きさと位置が決まったら**リターン**を押してください。

ここで次の節に対する準備のために，***F3***を押してStackウィンドウを閉じてください。CとPascalのどちらの例題プログラムをロードしたかにより，次のCの節か，その次のPascalの節に進んでください。

Turbo Debuggerの使い方(Cプログラムのデバッグ)

Moduleウィンドウの左端の桁(カラム)の矢印→(IBM PCでは右向きの三角)は，Turbo Debuggerがプログラムを停止させた位置を示します。まだ例題プログラムを実行していませんので，矢印とカーソルは次の実行可能な行にあります。

F7を押してください。ソースコードの1行が実行されて，矢印とカーソルが次の行に移動します。

Moduleウィンドウのタイトルの右端（ファイル名の隣り）を見てください。カーソルがある行の番号が表示されています。矢印キーを使ってカーソルを上下に移動させ，行番号がどのように変化するか見てください。

次にプログラムを実行して見ましょう。プログラムの実行を制御する方法は多数あります。**Run**メニューを開くと，そのリストを見ることができます。

ここでカーソルを行38に移動して，***F4***を押してください。プログラムが行38まで実行されます（行38は実行されません）。次に***F7***を押してください。行38が1行実行されます。行38は関数**showargs**の呼び出しです。その結果矢印は，関数**showargs**の先頭の行である行150にジャンプします。***F7***を押し続けると，関数**showargs**が1行ずつ実行され，**showargs**の実行が終わると行39（関数**showargs**を呼び出した行38の次の行）に戻ります。行38にいるときに***F7***のかわりに***F8***を押すと，矢印は関数**showargs**の先頭ではなく直接行39に移動します。***F8***は，ソースコードが1行ずつ実行される点は***F7***と同じですが，関数呼び出しがスキップ（一度に実行）される点が異なります。

図3.2　関数showargsから戻って停止した状態

```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TCDEMO  File: TCDEMO.C 39═══════════════════════════════════════════1┐
│          unsigned int  nlines, nwords, wordcount;                            │
│          unsigned long totalcharacters;                                      │
│                                                                              │
│          nlines = 0;                                                         │
│          nwords = 0;                                                         │
│          totalcharacters = 0;                                                │
│          showargs(argc, argv);                                               │
│►         while (readaline() != 0) {                                          │
│                  wordcount = makeintowords(buffer);                          │
│                  nwords += wordcount;                                        │
│                  totalcharacters += analyzewords(buffer);                    │
│                  nlines++;                                                   │
│          }                                                                   │
│          printstatistics(nlines, nwords, totalcharacters);                   │
│          return(0);                                                          │
│  }                                                                           │
│                                                                              │
│  /* make the buffer into a list of null-terminated words that end in         │
└──────────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────2┐
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
Alt: F2-Bkpt at F3-Mod F4-Anim F5-User F6-Undo F7-Instr F8-Rtn F9-To
```

プログラムをある場所まで実行するために，関数の名前または行の番号を直接指定することもできます。この方法によれば，ソースファイルの行をカーソルで指定して実行を開始させる必要はありません。***GRPH-F9***を押すと行のラベルを指定できます。***GRPH-F9***を押してみてください。するとプロンプトボックスが現われたはずです。readalineと入力して**リターン**を押してください。プログラムが実行され，関数**readaline**の先頭の行で停止します。

ブレークポイントを設定する

プログラムが停止する場所を制御するもう1つの方法は，ブレークポイントを使うことです。ブレークポイントを設定するための最も簡単な方法は，***F2***キーを使うことです。カーソルを行43に移動させて，***F2***を押してください。Turbo Debuggerは行43をハイライト表示にして，この行にブレークポイントが設定されていることを示します。

図3.3　行43に設定されたブレークポイント

```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TCDEMO  File: TCDEMO.C 41──────────────────────────────────────────────1┐
│         unsigned int  nlines, nwords, wordcount;
│         unsigned long totalcharacters;
│
│         nlines = 0;
│         nwords = 0;
│         totalcharacters = 0;
│         showargs(argc, argv);
│►        while (readaline() != 0) {
│                 wordcount = makeintowords(buffer);
│                 nwords += wordcount;
│                 totalcharacters += analyzewords(buffer);
│                 nlines++;
│         }
│         printstatistics(nlines, nwords, totalcharacters);
│         return(0);
│ }
│
│ /* make the buffer into a list of null-terminated words that end in
└────────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────────2┐
│
└────────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

ここで***F9***を押してください。***F9***はプログラムを自由に実行させます。すると，デバッガの画面から例題プログラムの画面に切り替わります。現在例題プログラムが動作しており，あなたがテキストを1行入力するのを待っています。abc,スペース,defとタイプして**リターン**を押してください。するとデバッガの画面に戻り，矢印はソースコードの行43にあるはずです。この行は，プログラムを停止させるブレークポイントを設定した行です。ブレークポイントについての詳しい説明は，第7章を参照してください。

Watchesウィンドウ

画面下部のWatchesウィンドウは，指定された変数の値を表示します。例として，変数*nwords*の値を見てみましょう。カーソルを行41の変数*nwords*に移動して***CTRL-W***を押してください。***CTRL-W***は，ローカルメニューコマンド***GRPH-F10 W***のショートカットです。

図3.4 Watchesウィンドウの中の変数

```
  File   View   Run   Breakpoints   Data   Window   Options                 READY
┌Module: TCDEMO  File: TCDEMO.C 39══════════════════════════════════════════1┐
│         nwords = 0;
│         totalcharacters = 0;
│         showargs(argc, argv);
│▶        while (readaline() != 0) {
│                 wordcount = makeintowords(buffer);
│                 nwords += wordcount;
│                 totalcharacters += analyzewords(buffer);
│                 nlines++;
│         }
│         printstatistics(nlines, nwords, totalcharacters);
│         return(0);
│ }
│
│ /* make the buffer into a list of null-terminated words that end in
│  * in two nulls, squish out white space
│  */
│ static int makeintowords(char *bufp) {
│         unsigned int nwords;
└────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────2┐
│nwords                          unsigned int 0 (0x0)                        │
└────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

すると上の図に示すように，Watchesウィンドウの中に変数*nwords*が現われ，その右側に***nwords***のデータ型（**unsigned int**）と値が表示されます。プログラムを実行すると値の表示が更新され，いつでも現在の値が表示されます。

Cの単純なデータを調べる

プログラムの実行を停止させたら，Inspectコマンドを使ってデータを調べてみましょう。Inspectコマンドを使うと，プログラムを書くときと同じ書式で，データ構造を調べることができます。Inspectコマンドは，多くのローカルメニューとDataメニューの中にあります。

それでは変数*nwords*の値を調べてみましょう。*nwords*の文字のどれかにカーソルを移動して，***CTRL-I***を押してください。次のようなInspectorウィンドウがポップアップします。

図3.5　Inspectorウィンドウの例

```
  File   View   Run   Breakpoints   Data   Window   Options            READY
Module: TCDEMO  File: TCDEMO.C 41 ─────────────────────────────────────1
         nwords = 0;
         totalcharacters = 0;
         showargs(argc, argv);
▶        while (readaline() != 0) {
                 wordcount = makeintowords(buffer);
                 nwords += wordcount;
                ┌Inspecting nwords──────────3┐fer);
                │Register                    │
         }      │unsigned int        0 (0x0) │
         printsta└───────────────────────────┘ters);
         return(0);
 }

 /* make the buffer into a list of null-terminated words that end in
  * in two nulls, squish out white space
  */
 static int makeintowords(char *bufp) {
         unsigned int nwords;
Watches ───────────────────────────────────────────────────────────────2
nwords                        unsigned int 0 (0x0)
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

上の図のInspectorウィンドウの中の最初の行（Register）は，コンパイラが，最適化によりこの変数をCPUレジスタに置くようにしたことを示します。でも変数がどこに格納されているか，気にする必要はありません。そういうことは，Turbo Debuggerがすべて引き受けてくれます。そのおかげで，すべてのデータを，ソースプログラムの中で指定するだけで見ることができます。第2行は，*nwords*にどの型のデータが格納されているかを示します。型の右に表示されているのは，その変数の現在の値です。

変数を調べたら，***ESC***を押してInsectorウィンドウを閉じてください。他のウィンドウとまったく同じように，***F3***を押して閉じることもできます。

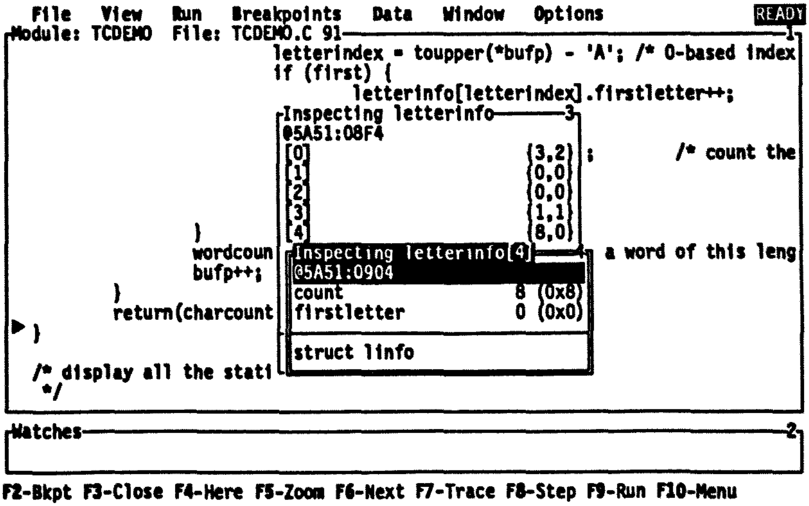
さてここでやったことを復習してみましょう。***CTRL***を使って，Moduleウィンドウのローカルメニューコマンドへのショートカットを使いました。次に***I***を押してInspectorコマンドを指定しました。

Moduleウィンドウの中に表示されていないデータを調べるには，***GRPH-D I***を押します。***GRPH-D I***を押しみてください。するとプロンプトボックスが現われて，調べたい変数の入力を求めます。letterinfoとタイプして**リターン**を押してください。すると，letterinfo配列の要素の値を表示するInspectorウィンドウが現われます。Inspectorウィンドウのタイトルは，表示されているデータの名前を示します。タイトルの下の行は，配列letterinfoの最初の要素のアドレスを示します。矢印キーを使ってウィンドウの中で表示をスクロールし，letterinfoを構成する26個の要素をすべて見てください。次の節で，この複合データを調べる方法を説明します。

Cの複合データを調べる

配列や構造体のような複合データは，複数の要素を含みます。配列*letterinfo*の4番目の要素（[3] により示される要素）に移動してください。次に***GRPH-F10***を押して，ローカルメニューを表示させ，***I***を押してInspectorコマンドを選択してください。すると新しいInspectorウィンドウが現われ，配列*letterinfo*の4番目の要素の内容が表示されます。

図3.6　Inspectorウィンドウの構造体の表示



要素名のひとつにカーソルを移動すると，Inspectorウィンドウの最下部のペインに，その要素のデータ型が表示されます。要素のひとつがまた複合データのときは，Inspectorコマンドを使って，その要素のデータ構造を調べることができます。

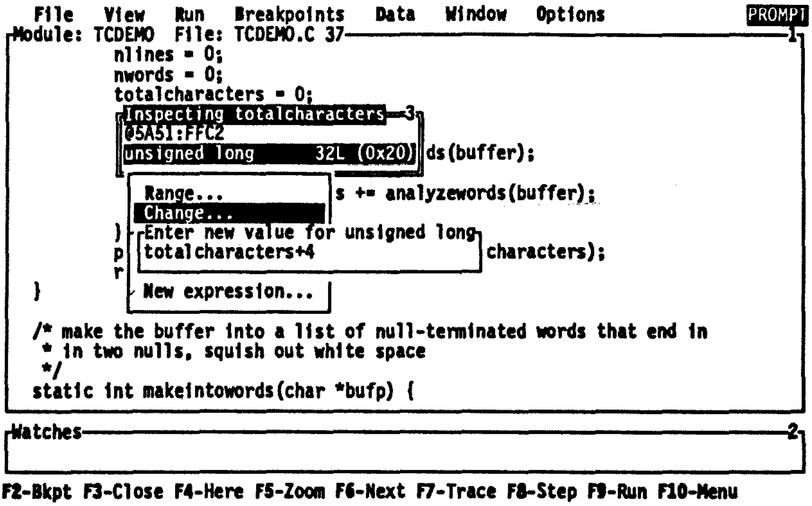
ここで***F3***を押して両方のInspectorウィンドウを閉じ，Moduleウィンドウに戻ってください（***F3***は複数のInspectorウィンドウを一度に閉じるのに便利な方法です。***ESC***は一番後に開かれたInspectorウィンドウだけを閉じます）。

Cのデータの値を変更する

これまで，プログラムの変数の値を調べる方法について説明しました。こんどはデータの値を変更してみましょう。

矢印キーを使って，ソースファイルの行37を表示してください。カーソルを変数*totalcharacters*に置き***CTRL-I***を押してください。Inspectorウィンドウが開いたら，***GRPH-F10***を押してローカルメニューを開いてください。***C***を押して，ローカルメニューからChangeコマンドを選択してください（Inspectorウィンドウから***CTRL-C***を押すとショートカットになります）。するとプロンプトが現われて，新しい値の入力を待ちます。

図3.7　Changeコマンドのプロンプト



演算結果が数値になる，任意のCの式を入力することができます。totalcharacters+4と入力して**リターン**を押してください。するとInspectorウィンドウの中の値が，新しい値10L（0xA）に変わるはずです。

Moduleウィンドウに表示されていないデータ項の値を変更するには，***GRPH-D E***を押します。***GRPH-D E***を押してみてください。するとプロンプトボックスが現われて，値を変更しようとする変数名の入力を待ちます。ここでargcと入力して**リターン**を押し，さらに***TAB***キーを2回押して，New expressionというラベルの付いている行に移動してください。

それから123と入力して**リターン**を押してください。argcの値が123 (0x7B) に変るはずです。

Turbo DebuggerによるCプログラムのデバッグの簡単な説明はこれで終わりです。例題プログラムを用いた，より詳しい使い方の説明は，第13章のデバッグセッションの例を参照してください。第13章では，ここで使った例題プログラムにバグを入れたものを，例題プログラムとして使っています。

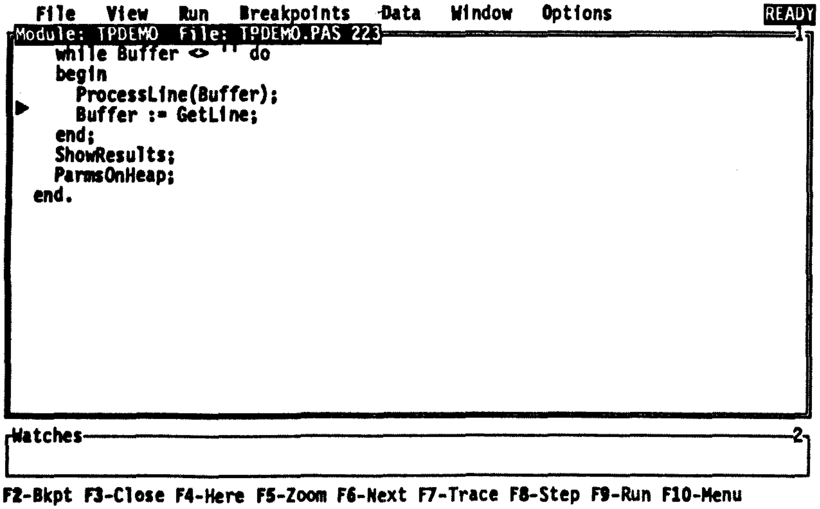
Moduleウィンドウの左端の桁（カラム）の矢印（→）は，Turbo Debuggerがプログラムを停止させた位置を示します。まだ例題プログラムを実行していませんので，矢印はプログラムの先頭にあります。

F7を押してください。ソースコードの1行が実行されて，矢印とカーソルが次の行に移動します。

Moduleウィンドウのタイトルの右端をみてください。カーソルがある行の番号が表示されています。矢印キーを使ってカーソルを上下に移動させ，行番号がどのように変化するか見てください。

カーソルを行222に移動して，***F4***を押してください。プログラムが行222まで実行されます(行222は実行されません)。するとTPDEMOは，あなたに文字列を入力するように求めます。何か2,3の文字を入力して**リターン**を押してください。ここでカーソルを行222に置いたままF7を押して，ソースコードをもう1行実行してください。いま実行した行は別の手続きの呼び出しですので，矢印は呼び出された手続き*ProcessLine*の最初の行に現われます。次に***GRPH-F8***を押してください。***GRPH-F8***を押すと*ProcessLine*が実行され，呼び出し側に戻ったときに実行が停止します。このコマンドは，関数や手続きを素早く通り抜けたいときに非常に便利です。

図3.8　手続きから戻って停止した状態



プログラムをある場所まで実行するために，関数（手続き）の名前または行の番号を直接指定することもできます。この方法をとれば，ソースファイルの行をカーソルで指定して実行を開始させる必要はありません。***GRPH-F9***を押すと，行のラベルを指定することができます。***GRPH-F9***を押してみてください。プロンプトボックスが現われたら，***GetLine***と入力して**リターン**を押してください。プログラムが実行されて，関数***GetLine***の先頭の行で停止します。

ブレークポイントを設定する

プログラムが停止する場所を制御するもう1つの方法は，ブレークポイントを使うことです。ブレークポイントを設定する最も簡単な方法は，***F2***キーによる設定です。カーソルを行120に移動させて，***F2***を押してください。Turbo Debuggerは行120をハイライト表示にして，この行にブレークポイントが設定されていることを示します。

図3.9　行120に設定されたブレークポイント

```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: TPDEMO  File: TPDEMO.PAS 120──────────────────────────────────────1┐
│    i : Integer;
│    WordLen : Word;
│
│  begin { ProcessLine }
│    Inc(NumLines);
│    i := 1;
│    while i <= Length(S) do
│    begin
│      { Skip non-letters }
│      while (i <= Length(S)) and not IsLetter(S[i]) do
│        Inc(i);
│
│      { Find end of word, bump letter & word counters }
│      WordLen := 0;
│      while (i <= Length(S)) and IsLetter(S[i]) do
│      begin
│        Inc(NumLetters);
│        Inc(LetterTable[UpCase(S[i])].Count);
└────────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────────2┐
│
└────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

ここで***F9***を押してください。***F9***はプログラムを自由に実行させます。するとデバッガの画面から例題プログラムの画面に切り替わります。現在例題プログラムが動作しており，

あなたがテキストを1行入力するのを待っています。abc＜スペース＞defと入力して**リターン**を押してください。するとデバッガの画面に戻り，矢印はソースコードの行120に移動しているはずです。この行は，プログラムを停止させるブレークポイントを設定した行です。ブレークポイントについての詳しい説明は第7章を参照してください。

Watchesウィンドウ

画面下部のWatchesウィンドウには，指定された変数の値が表示されます。例として変数*NumWords*の値を見てみましょう。カーソルを行143の変数*NumWords*に移動して***CTRL-W***を押してください。***CTRL-W***は，ローカルメニューコマンド***GRPH-F10 W***のショートカットです。

図3.10　Watchesウィンドウの中の変数

```
  File   View   Run   Breakpoints   Data   Window   Options                READY
 Module: TPDEMO  File: TPDEMO.PAS 143 ─────────────────────────────────────1─
        Inc(LetterTable[UpCase(S[i])].Count);
        if WordLen = 0 then                    { bump counter }
          Inc(LetterTable[UpCase(S[i])].FirstLetter);
        Inc(i);
        Inc(WordLen);
      end;

      { Bump word count info }
      if WordLen > 0 then
      begin
        Inc(NumWords);
        if WordLen <= MaxWordLen then
          Inc(WordLenTable[WordLen]);
      end;
    end; { while }
  end; { ProcessLine }

  function GetLine : BufferStr;
 Watches ───────────────────────────────────────────────────────────────────2─
 NumWords                     2 ($2) : WORD
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

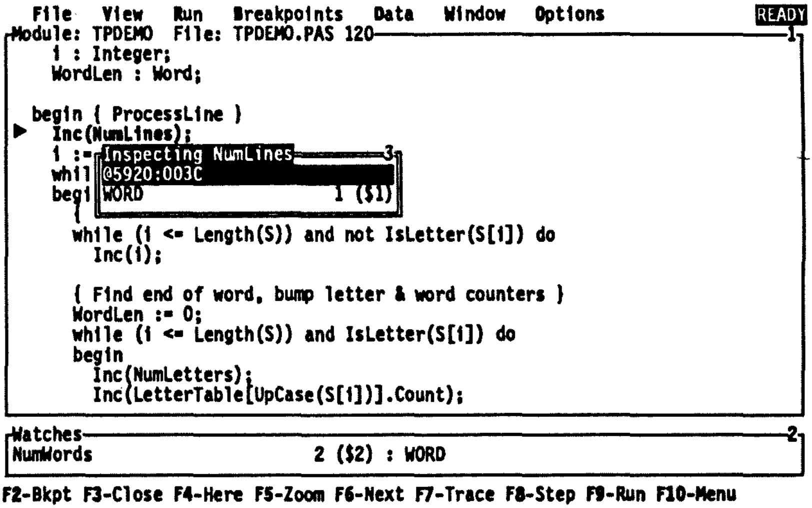
すると上の図のように，Watchesウィンドウの中に変数*NumWords*が現われます。その右側に，*NumWords*のデータ型（word）と値が表示されます。プログラムを実行すると，値の表示が更新され，いつでも現在の値が表示されます。

Pascalの単純なデータを調べる

プログラムの実行を停止させたら，Inspectコマンドを使ってデータを調べてみましょう。Inspectコマンドを使うと，プログラムを書くときと同じ書式で，データ構造を調べることができます。Inspectコマンドは，多くのローカルメニューとDataメニューの中にあります。

それでは変数*NumLines*の値を調べてみましょう。*NumLines*の文字のどれかにカーソルを移動して，***CTRL-I***を押してください。次のようなInspectorウィンドウがポップアップします。

図3.11　Inspectorウィンドウ



Inspectorウィンドウの中の最初の行は，変数の名前を示します。2番目の行は，変数*NumLines*のアドレスを示します。3番目の行は，*NumLines*に格納されているデータの型（Pascalのword型）と，現在の値を示します。

変数を調べたら，***ESC***を押してInspectorウィンドウを閉じてください。他のウィンドウと同じように，***F3***を押して閉じることもできます。

さてここでやったことを復習してみましょう。***CTRL***を押して，Moduleウィンドウのロ

ーカルメニューコマンドへのショートカットを利用しました。次に**I**を押してInspectコマンドを指定しました。

Moduleウィンドウに表示されていないデータを調べるには, ***GRPH-D I***を押します。***GRPH-D I***を押しみてください。するとプロンプトボックスが現われて, 調べたい変数の入力を求めます。LetterTableと入力して**リターン**を押してください。すると*LetterTable*の値を表示するInspectorウィンドウが現われます。矢印キーを使ってウィンドウの中の表示をスクロールし, *LetterTable*を構成する26個の要素をすべて見てください。次の節で, この複合データを調べる方法について説明します。

Pascalの複合データを調べる

配列や構造体のような複合データは, 複数の要素を含みます。配列*LetterTable*の4番目の要素(['D']により示される要素)に移動してください。次に***GRPH-F10***を押して, Inspectorウィンドウのローカルメニューを表示させてください。さらに**I**を押して, Inspectコマンドを選択してください。すると新しいInspectorウィンドウが現われて, 配列*LetterTable*の4番目の要素が表示されます。

図3.12 Inspectorウィンドウの構造体の表示

```
  File  View  Run  Breakpoints  Data  Window  Options                  READY
┌Module: TPDEMO  File: TPDEMO.PAS 120────────────────────────────────────1┐
│   i : Integer;                                                          │
│   WordLen : Word;                  ┌Inspecting LetterTable──────3┐       │
│                                    │@5920:0058                   │       │
│  begin { ProcessLine }             │['A']              (1,1)     │       │
│►  Inc(NumLines);                   │['B']              (1,0)     │       │
│   i := 1;                          │['C']              (1,0)     │       │
│   while i <= Length(S) do          │['D']              (1,1)     │       │
│   begin                            ├┌Inspecting LetterTable['D']──4┐     │
│     { Skip non-letters }           ││@5920:0064                    │     │
│     while (i <= Length(S)) and not└│COUNT                1 ($1)   │     │
│       Inc(i);                       │FIRSTLETTER          1 ($1)   │     │
│                                     ├──────────────────────────────┤     │
│     { Find end of word, bump letter│record LINFOREC               │     │
│     WordLen := 0;                  └──────────────────────────────┘     │
│     while (i <= Length(S)) and IsLetter(S[i]) do                        │
│     begin                                                               │
│       Inc(NumLetters);                                                  │
│       Inc(LetterTable[UpCase(S[i])].Count);                             │
└─────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────2┐
│NumWords                      2 ($2) : WORD                             │
└────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

要素名のひとつにカーソルを移動すると, そのInspectorウィンドウの最下部のペイン

に，その要素のデータ型が表示されます。これらの要素のひとつがまた複合データのときは，Inspectコマンドを使って，その要素のデータ構造を調べることができます。

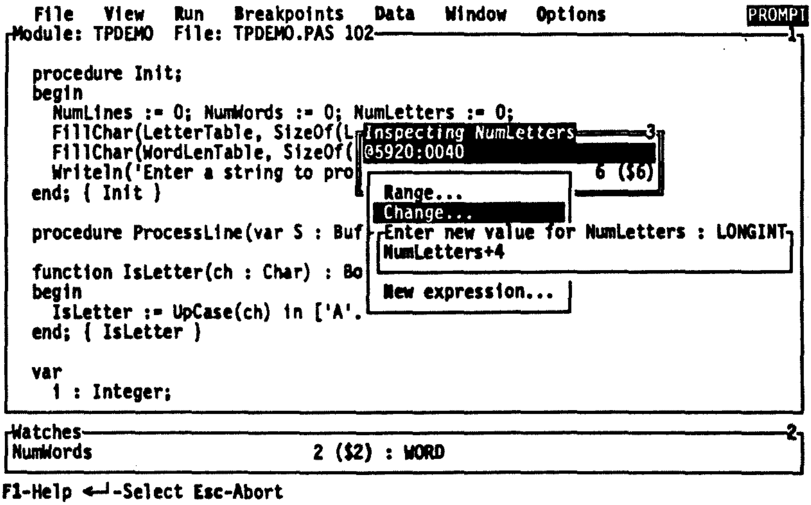
ここで***F3***を押して両方のインスペクタウィンドウを閉じ，Moduleウィンドウに戻ってください（***F3***は，複数のInspectorインスペクタウィンドウを一度に閉じるのに便利な方法です。***ESC***は，一番上のInspectorウィンドウだけを閉じます）。

Pascalのデータの値を変更する

これまでプログラムの変数の値を調べる方法について説明しました。ここではデータの値を変更してみましょう。

矢印キーを使って，ソースファイルの行102を表示してください。カーソルを変数*NumLetters*に置き，***CTRL-I***を押してください。Inspectorウィンドウが開いたら，***GRPH-F10***を押してローカルメニューを開いてください。次に***C***を押して，ローカルメニューからChangeコマンドを選択してください（Inspectorウィンドウから***CTRL-C***を使うこともできます）。するとプロンプトが現われて，新しい値の入力を求めてきます。

図3.13　Changeコマンドのプロンプト



Changeコマンドでは，演算結果が数値になる任意のPascalの式を入力することができます。NumLetters＋4とタイプして，**リターン**を押してください。するとInspectorウィンド

ウの中の値が，新しい値10に変わります。

Moduleウィンドウに表示されていないデータの値を変更するには，***GRPH-D E***を押します。***GRPH-D E***を押してみてください。するとプロンプトボックスが現われて，値を変更しようとする変数名の入力を求めてきます。ここではNumLinesと入力して**リターン**を押してください。すると変数*NumLines*が中間のペインに表示されます。***TAB***を2度押してから，123と入力して**リターン**を押してください。変数*NumLines*の値が123にセットされるはずです。

Turbo DebuggerによるPascalプログラムのデバッグの簡単な説明は，これで終わりです。例題プログラムを用いた，もっと詳しい使い方の説明は，第13章のサンプルデバッグセッションを読んでください。第13章では，ここで使った例題プログラムにバグを入れたものを，例題プログラムとして使っています。

第4章

Turbo Debuggerの起動

Turbo Debuggerでプログラムをデバッグするには，TDと入力し，その後にデバッグするプログラムの名前を入力して**リターン**を押します。そうするとTurbo Debuggerが起動して，あなたが指定したプログラムをロードし，ソースコードを画面のウィンドウに表示して，デバッグを開始する準備を整えてくれます。

DOSのプロンプトからあなたのプログラムを実行して，それを使っているうちにバグを見つけたときは，まずあなたのプログラムを終了して，Turbo Debuggerのもとにロードし直さなければなりません。

この章では，まずプログラムをデバッグできる準備するための方法について説明します。次にコマンドライン引数やOptionsメニューのコマンドを使って，デバッグするプログラムに適したデバッグ環境を設定する方法，および設定したオプションをコンフィギュレーションファイルに保存する方法を説明します。さらにTurbo Debuggerの中からDOSのコマンドを実行する方法を説明します。最後にデバッグを終了してDOSに戻る方法について説明します。

プログラムをデバッグできるように準備する

ボーランドのTurboファミリィ言語で書かれたプログラムをコンパイルしリンクするときに，コンパイラに必要なすべてのデバッグ情報を生成するように指示しなければなりません。あなたのプログラムのオブジェクトモジュールを，デバッグ情報を生成せずにコンパイルしたときは，プログラム全体をソースコードでデバッグできるようにするために，プログラムのすべてのモジュールを再コンパイルする必要があります。プログラムの1部のモジュールに対してだけデバッグ情報を生成することもできますが，混乱を引き起こす恐れがあります。拡張メモリ(EMS)を使っているとき，メモリスペースを必要とするとき，いくつかのモジュールのコードにはバグがないことが確実なとき以外は，すべてのモジュールを再コンパイルすることをお奨めします。

Turbo Cプログラムの準備

Turbo C統合開発環境(TC)を使用しているときは，ソースプログラムをコンパイルする前に，**Debug/Source debugging**オプションをOnまたはStandaloneに，さらに**Options/Compiler/Code generation/OBJ debug information**をOnに設定してください。

スタンドアロンコンパイラ(TCC)を使用しているときは，**-v**コマンドラインオプションを指定してください。

TLINKをスタンドアロンリンカとして使用しているときは，.EXEファイルの最後にデバッグ情報を付加するために，/**v**オプションを使用してください。

最適化(オプティマイズ)を抑制したいときは，**-O**オプションを指定しないか，**-O**オプションを明示的に(どこかで指定されている場合にも確実に)オフにするために**-O-**を指定してください。最適化をオフにすると，ソースコードを1行ずつ実行しているときに，複数行のソースコードがスキップされるように見える動作がなくなります。

Turbo Pascalプログラムの準備

Turbo Pascalのバージョン5.0またはそれ以降のバージョンが必要です。それ以前のバージョンには，Turbo Debuggerが使用できるようにデバッグ情報を.EXEファイルに組み入れる機能がありません。

統合開発環境(TURBO.EXE)を使用されているときは，**Debug**メニューの中の**Standalone debugging**の設定をOnに変更してください。さらに，**Options/Compiler/Debug information**

をOnにしてください。ローカルシンボル(手続きおよび関数の内部で宣言されたシンボル)を参照できるようにしたいときは，Options/Compiler/Local symbolsの選択をOnにするか，プログラムの先頭に｛$L+｝コンパイラ指令を置いてください。これでプログラムをコンパイルできます。

コマンドライン版（TPC.EXE）を使用されているときは，/vコマンドラインオプションを使ってコンパイルしてください。デバッグ情報とローカルシンボル情報の作成は，デフォルトでオンになっています。必要でないときは，/$コマンドラインオプションでオフにすることができます。

Turbo Assemblerプログラムの準備

Turbo Assemblerを使用しているときは，すべての必要なデバッグ情報を生成するために，**-zi**コマンドラインオプションを指定してください。

プログラムをTLINKを使ってリンクするときは，.EXEファイルの終わりにデバッグ情報を付加するために/**v**オプションを指定してください。

マイクロソフト社の言語で書かれたプログラムの準備

CodeViewでデバッグ可能なプログラムは，ユーティリティプログラムTDCONVRT.EXEによりTurbo Debuggerのフォーマットに変換することができます。詳しい説明は，このマニュアルの付録Bを参照してください。

Turbo Debuggerの起動

Turbo Debuggerを起動するには，DOSのプロンプトのもとでTDと入力し，その後に必要なコマンドライン引数を入力して**リターン**を押します。

Turbo Debuggerを起動するときは，メニューシステムの入っているファイルTD.OVLが必要です。Turbo Debuggerは，起動時にTD.OVLが見つからないと，メッセージを表示してそのことを知らせます。このメッセージが表示されたら，Turbo Debuggerと同じディレクトリにTD.OVLを入れてもう一度起動してください。

コマンドライン引数には，デバッガオプションやデバッグするプログラムの名前などを指定することができます。

ただTDと入力して**リターン**を押すと，Turbo Debuggerが起動し，デフォルトのオプションを使用します。

コマンドラインの一般的な書式は，次のとおりです。

```
TD [options] [progname [progargs]]
```

カッコ([])で囲まれた項目は，省略することができます。カッコは入力する必要はありません。*Progname*は，デバッグしようとするプログラムの名前です。プログラム名の後には，そのプログラムの引数を指定することができます。次にコマンドライン引数の例を示します。

コマンド	**意味**
td -sc progl a b	**-sc**オプションを指定してデバッガを起動し，2つのコマンドライン引数*a*, *b*を与えてプログラム*progl*をロードします。
td prog2 -x	デフォルトオプションを使ってデバッガを起動し，引数*-x*を与えてプログラムをロードします。

Turbo Debuggerでデバッグできるボーランドの製品は，Turbo Pascalのバージョン5.0またはそれ以降のバージョン，Turbo Cのバージョン2.0またはそれ以降のバージョン，またはTurbo Assemblerのバージョン1.0またはそれ以降のバージョンです。

Turbo Debuggerでデバッグする前に，ソースコードを，すべてのデバッグ情報の生成をオンにして，実行可能ファイルにコンパイルしておかなければなりません。

またTurbo Debuggerを起動するときに，.EXEファイルと元のソースファイルの両方が

必要なことに注意してください。Turbo Debuggerは，ソースファイルを見つけるために，次のディレクトリを探します。まずコンパイラがコンパイルしたときにソースファイルを見つけたディレクトリ，次にOptionsメニューのPath for sourceコマンドで指定されているディレクトリ，その次にカレントディレクトリ，最後に.EXEファイルが入っているディレクトリです。

コマンドラインオプション

すべてのコマンドラインオプションは，頭にマイナス記号(-)をつけ，TDコマンド，他のオプション，ファイル名と，少なくとも1つの空白で区切らなければなりません。オプションの後にマイナス記号をつけると，そのオプションをオフにすることができます。たとえば-**vg**-とすると，グラフィックスのセーブがオフになります。コンフィギュレーションファイルの中でオンにされているオプションがあるときは，この機能を使ってオフにすることができます。また付録Eに説明されているTDINSTコンフィギュレーションプログラムを使って，コンフィギュレーションファイルを変更することもできます。

次にすべてのコマンドラインオプションを説明します。またこれらのコマンドラインオプションのリストを付録Aに掲載します。

-cオプション

指定されたコンフィギュレーションファイルから，オプションの設定を読み込みます。このオプションを指定しないときは，（もしあれば）TDCONFIG.TDの設定が使用されます。たとえば

```
TD -cMYCONF.TD TCDEMO
```

と指定したときは，コンフィギュレーションファイルMYCONF.TDからオプションの設定を読み込み，プログラムファイルTCDEMOをロードします。-**c**とプログラムファイル名の間には，空白を入れてはいけません。

-dオプション

-**d**オプションは，デバッガの表示に影響を及ぼします。

-do

第2のディスプレイにデバッガの画面を表示します。第1のディスプレイでユーザプログラムの画面表示を見ながら，第2のディスプレイでデバッグすることができます。

このオプションはIBM PCでのみ使用可能です。PC-9801では常に無視されます。

-dp

1つのディスプレイページでデバッガの画面を表示し，もう1つのディスプレイページでデバッグしているプログラムの画面を表示します。このオプションにより，2つの画面の切り替えにかかる時間を短縮することができます。IBM PCの場合には，これはカラーディスプレイのためのデフォルトオプションです。IBM PCではこのオプションは，カラーディスプレイでなければ使えません。なぜなら，複数のディスプレイページを持つのは，カラーディスプレイだけだからです。デバッグしているプログラム自体が複数のディスプレイページを使うときは，このオプションは使えません。

-ds

デバッグしているプログラムが実行されるたびに，またデバッガに制御が戻るたびに，メモリから画面全体をロードしてデバッガの画面とプログラムの画面を切り替えます。この方法は，2つの画面イメージを切り替え表示するのに最も時間のかかる方法です。そのかわり，どのようなディスプレイハードウェアに対しても，またディスプレイに特殊なことをするプログラムに対しても動作します。IBM PCでは，モノクローム(白黒)ディスプレイのためのデフォルトオプションです。

-gオプション(PC-9801のみ)

-gオプションは，Turbo Debuggerの46行表示モードに関する指定を行ないます。**-g**または**-g**＋を指定すると，46行モードが使用可能になります。**-g**-とすると，46行モードは使用不可になります。

-gN

46行モードのために使用するGVRAMのバンクを，0または1で指定します。GVRAMが1画面分しかないマシンや，GVRAMをRAMディスクとして使用している場合には，**-g0**を指定する必要があります。

-hと-?オプション

コマンドラインオプションの意味と書式を示すヘルプ画面を表示します。

-iオプション

プロセスIDの切り替えをオンにします。DOSの内部でデバッグしているときやDOSのシステムコールがアクティブのときは，このオプションを使用しないでください。このオプションの詳細については付録Cを参照してください。ほとんどのプログラムのデバッグでは，このオプションを気にする必要はありません。

-lオプション

アセンブラモードで起動し，CPUウィンドウを表示します。コンパイラのスタートアップコードは実行しません。

-mオプション

ヒープ領域の大きさを*n*Kバイトに設定します。書式は次のとおりです。

```
-m n
```

デフォルトの大きさは40Kです。上限は64Kです。メモリを必要とするときは，このオプションを使ってTurbo Debuggerが使用するヒープ領域を縮小してください。小さいプログラムをデバッグするときは，このオプションを使ってヒープ領域を拡大してください。Turbo Debuggerは，このヒープ領域に，ヒストリリストなどの一時的な情報を格納します。

```
TD -m64 TCDEMO
```

-rオプション

-rオプションは，リモートデバッグリンクに影響を及ぼします。

-rpN

リモートリンクポートをポート*N*に設定します。*N*のとり得る値は1または2です。1はCOM1を示し，2はCOM2を示します。

-rsN

リモートリンクの速度を設定します。*N*のとり得る値は1または2(IBM PCでは1から3)です。1は9,600bps，2は40/31Kbps(IBM PCでは40Kbps)，3は115Kbpsを示します。デフォルトは40/31Kbps(IBM PCでは115Kbps)です。

-sオプション

-sオプションは，Turbo Debuggerがソースコードやプログラムのシンボルを扱う方法に影響を及ぼします。

-sc

シンボル名の文字について，大文字小文字の区別をしません。プログラムが大文字小文字の区別をオンにして(リンカのオプションで指定)リンクされている場合にも，Turbo Debuggerは文字の大小を無視します。

-scオプションを指定しないときは，Turbo Debuggerは，プログラムが大文字小文字の区別をオフにしてリンクされているときだけ，文字の大小を無視します。

注意：このオプションは，Turbo Pascalのプログラムをデバッグするときは無効です(Turbo Pascalは，文字の大小を区別しないからです)。

-sd

Turbo Debuggerがソースファイルを見つけるために探すべきディレクトリを指定します。書式は次のとおりです。

```
-sddirname
```

1つの**-sd**オプションには1つのディレクトリしか指定できません。複数のディレクトリを指定するには**-sd**オプションを繰り返し使用してください。Turbo Debuggerは，指定されたディレクトリを，指定された順に探します。*dirname*は相対パス，絶対パスのどちらでもかまいません。またディスクドライブ名を含めることもできます。コンフィギュレーションファイルにディレクトリを指定している場合には，そのリストの後に，このオプションで指定したディレクトリが追加されます。

-vオプション(IBM PCのみ)

-vオプションは, Turbo Debuggerがビデオハードウェアを操作する方法に影響を及ぼします。

-vg

プログラムの画面に完全なグラフィクスイメージを表示します。8Kバイトのメモリが余計に必要ですが，グラフィックス表示モードを使用するプログラムもデバッグすることができます。Turbo Debuggerのもとで実行したときプログラムのグラフィクス画面がおかしい場合には，このオプションを設定してみてください。

-vn

43/50行モードグラフィックの表示ができなくなります。このオプションを指定すると，メモリが少し節約できます。EGAまたはVGAを使用していて，いったんTurbo Debuggerを起動したら43行モードにも50行モードにも切り替えないことがわかっている場合には，このオプションを使用してください。

コンフィギュレーションファイル

Turbo Debuggerには，コマンドラインオプションのデフォルトの設定が組み込まれています。コンフィギュレーションファイルを作成すると，デフォルトの設定を上書きすることができます。TDINSTインストールプログラムを使うと，組み込みのデフォルトの設定を変更することができます。またコンフィギュレーションファイルを作成することもできます。

Turbo Debuggerは，コンフィギュレーションファイルTDCONFIG.TDを見つけるために，最初にカレントディレクトリ，2番目にTDINSTにより作成されたTURBOディレクトリ，3番目にTD.EXEの入っているディレクトリを探します。DOSのバージョン2.1を使用している場合には，TD.EXEの入っているディレクトリは探しません。

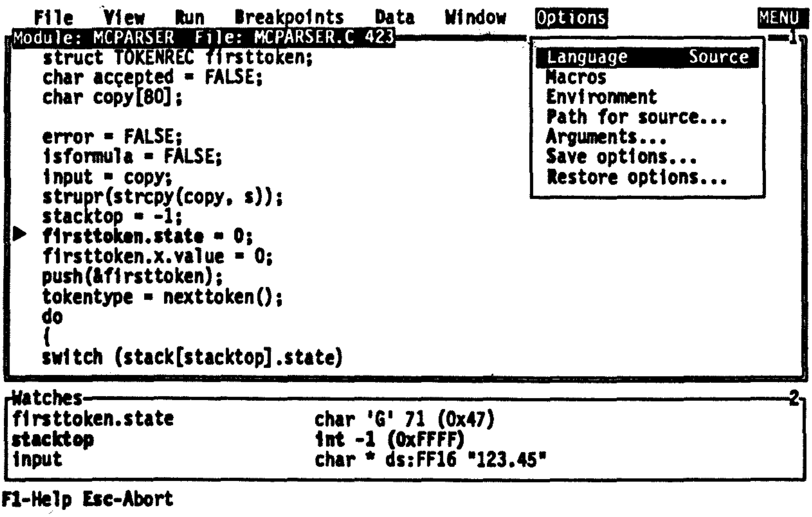
Turbo Debuggerがコンフィギュレーションファイルを見つけたときは，コンフィギュレーションファイルの設定がデフォルトの設定を上書きします。またTurbo Debuggerを起動するときに指定したコマンドラインオプションは，デフォルトの設定およびコンフィギュレーションファイルの設定を上書きします。

付録Eに，インストールプログラムを使ってコンフィギュレーションファイルを作成する方法が説明されています。

Optionsメニュー

Optionsメニューにより，Turbo Debuggerの画面や動作を制御する多数のパラメータを設定することができます。以下の節では，各メニューコマンドについて説明します。

図4.1　Optionsメニュー



Languageコマンド

式の記述に使用する言語を選択します。各言語で使用できる演算子や評価規則については，第9章に説明されています。

Macrosコマンド

このコマンドは，新しいキーストロークマクロを設定したり，すでに設定されているキ

ーストロークマクロを削除することができるもう1つのメニューを表示します。

Create

キーストロークの記録を開始します。まずキーストロークマクロを設定したいキーの入力を求めます。マクロを設定したいキーを押してください。

次に記録したいキーストロークを入力することができます。記録セッションの進行中は，右上隅に"Recording"というメッセージが表示されます。マクロの記録中も，キーの機能は，それ以外のときと同じです。

キーストロークの記録が終わったら，***F10***/Options/Macros/Stop recordingを選択するか，そのショートカットの***GRPH-－***を押してください。

GRPH-^ (IBM PCでは***GRPH-＝***)は，Createコマンドのショートカットです。

Stop recording

キーに設定されるキーストロークの記録を終了します。***F10***/Options/Macros/Createコマンドでキーストロークマクロの設定を開始したときは，このコマンドで終了してください。

GRPH-－は，このコマンドのショートカットです。

Remove

キーに設定されているマクロを削除します。設定されているマクロを解除したいキーの入力を求めてきます。

Delete all

すべてのキーストロークマクロの設定を解除し，各キーをもとの機能に戻します。

Environmentコマンド

このコマンドは，Turbo Debuggerの画面表示を制御することができるオプションのメニューを表示します。このメニューには，次のようなオプションがあります。

Integer format

このオプションは，整数を表示するための3つの表示形式のトグル(切り換えスイッチ)になっています。

Decimal 整数を通常の10進数で表示します。

Hex 整数を16進数で，選択されている言語の表記方法で表示します。

Both 整数を10進数と16進数の両方で表示します。16進数は10進数の後にカッコで囲んで表示されます。

Display swapping

Turbo Debuggerの画面からデバッグしているプログラムの画面への切り替え方法を指定します。次の3つの方法に順番に切り替わるトグルになっています。

None 2つの画面の切り替えを行ないません。ディスプレイに何も出力しないプログラムをデバッグするときは，このオプションを使ってください。

Smart プログラムから出力があったときだけ，ユーザ画面に切り替えます。Turbo Debuggerは，ある一定数のステップが実行されたとき，ビデオRAMに対して読み込み，書き込みを行なうアセンブリ命令やソースコードが実行されたときに画面を切り替えます。この設定がデフォルトの設定です。

Always プログラムが実行されるたびにユーザ画面に切り替えます。Smartオプションでは，プログラムのディスプレイへの出力を完全に捉えることができない場合には，このオプションを使ってください。このオプションを選択したときは，プログラムを実行するたびに画面がちらつきます。Turbo Debuggerの画面が，短時間，デバッグしているプログラムの画面に切り替わるからです。

Screen size

PC-9801：

Turbo Debuggerの画面に，テキスト画面による25行表示（Standard 25 line）と，グラフィック画面による46行表示（**High** density 46 line）のどちらを使用するかを設定しま

す。

IBM PC：

Turbo Debuggerの画面が，通常の25行表示と，EGAやVGA(ディスプレイアダプタ)を付けたときに可能な43行または50行表示のどちらを使用するかを設定します。

Tab size

タブ位置の間隔を設定します。タブによりインデントされて画面からはみ出しているコードのあるソースファイルは，タブ間隔を狭くすることにより，より多くのテキストを見ることができます。タブ間隔は，1から32の範囲で設定することができます。

Path for sourceコマンド

Turbo Debuggerが，ソースファイルを探すディレクトリを設定します。このオプションについての詳しい説明は，第8章のModuleウィンドウの説明を参照してください。

Argumentsコマンド

デバッグするプログラムに適したデバッグ環境を設定するための，新しいコマンドライン引数を設定することができます。第5章に詳しい説明があります。

Save optionsコマンド

現在のオプションの設定を，ディスクのコンフィギュレーションファイルに保存します。次の3通りの方法で保存することができます。

All　すべての設定(ウィンドウの配置とマクロの両方)を保存します。
Layout　ウィンドウの配置の設定だけを保存します。
Macros　マクロの設定だけを保存します。

名前の異なる複数のコンフィギュレーションファイルに，異なる設定を保存しておくこ

ともできます。

Restore optionsコマンド

Save optionsコマンドで保存したオプションの設定を，ディスクファイルから読み出します。複数のコンフィギュレーションファイルがあるときは，どのファイルを読み込むか指定しなければなりません。

Hardware statusコマンド（PC-9801のみ）

このコマンドは，グラフィックス画面やカーソルなどのハードウェアリソースに関する設定を行なうもう1つのメニューを表示します。このメニューの中で設定または変更可能な内容は，25行モードのみを使用している限りは，TD.EXE起動時の状態，およびユーザプログラムによって設定された状態を，Turbo Debuggerはいっさい変更しません。

Visual user G-page

ユーザグラフィックス画面の表示がオンになっている場合に，GVRAMのバンク＃0と＃1のどちらを表示するかを指定します。ユーザグラフィックス画面のオン/オフは，同じメニューのGraphics swtichによって切り替えることができます。

Active user G-page

デバッガからGVRAMにアクセスする場合(たとえばCPUウィンドウのDataペインで内容を見るとき）に，GVRAMのバンク＃0と＃1のどちらを対象とするかを指定します。

Graphics switch

ユーザグラフィックス画面を常に表示しておく(Enable)か，非表示にする(Disable)かを指定します。Enable(表示)とすると，グラフィックス画面は25行モードのデバッガ画面には表示されたままになります。Disable(非表示)とすると，（ユーザプログラム側でオンにしない限り）グラフィックス画面はオフのままになります。

Cursor switch

カーソルをどのように扱うかを指定します。デフォルトではDOSになっており，この場合はMS-DOSが保持している状態に合わせて表示/非表示を決定します。ただし，ユーザプログラムがハードウェアを直接操作してカーソルを制御している場合には，Turbo Debuggerにはカーソルの現在の状態を検出することはできません。Onを指定した場合には，Turbo Debuggerはカーソルを常に表示し，Offの場合には常に非表示にします。

EMS/VRAM switch

このコマンドは，EMSメモリのセグメントがB000(GVRAMセグメントと同じアドレス)から始まるPC-9801RA/RLなどのマシン（および同じ形式のメモリボードを装着している場合)のために用意されています。デバッガからそのメモリ領域にアクセスする際に，EMSとGVRAMのどちらを有効にするかを切り替えます。対象となるハードウェア以外の場合には，このコマンドを選択することはできません。

Turbo Debuggerの内部からDOSのコマンドを実行する方法

プログラムをデバッグしているとき，別のプログラムやユーティリティが必要になることがあります。このような場合には，**File/OS shell**を使うと，一時的にTurbo Debuggerから出て，DOSのプロンプトに移ることができます。

DOSのコマンドプロセッサに移るとき，デバッグしているプログラムが必要に応じてディスクに退避されます。そのため，利用できるメモリの全部を使うような大きなプログラムをデバッグしている途中でも，DOSのコマンドを実行することができます。もちろん，プログラムをディスクに退避したり，ディスクから再ロードするために，数秒かかることがありますが。

DOSのコマンドを実行したあと再びデバッグセッションに戻るには，EXITと入力して**リターン**を押してください。

DOSに戻る方法

GRPH-Xを押すと，いつでも，デバッグセッションを終了してDOSに戻ることができます。**File/Quit**を選択することもできます。

デバッグしていたプログラムに最初に割り当てられていたメモリは，すべて解放されます。またデバッグしていたプログラムが，DOSのブロックメモリアロケーションルーチンを使ってメモリを割り当てていたときは，そのメモリも解放されます。

第5章

プログラムの実行を制御する

プログラムをデバッグするときは，通常プログラムのある部分を実行して停止させ，その部分が正しく動作するかどうかを調べます。Turbo Debuggerには，プログラムの実行を制御する多数の方法が用意されています。

■機械語命令を1命令ずつ，またはソースコードを1行ずつ実行する。
■関数や手続きの呼び出しをスキップする。
■連続的にトレースする。
■現在の関数または手続きから呼び出し元(caller)に戻るまで実行する。
■指定した位置まで実行する。
■ブレークポイントに遭遇するまで実行する。

デバッグセッションは，デバッグしているプログラムが実行されている期間と，デバッガが動作している期間とからなります。デバッガが動作しているときは，Runメニューコマンドの1つを選択するか，それに対応するホットキーを押すことにより，プログラムを実行することができます。あなたのプログラムが実行されているときは，プログラムの指定された部分の実行が終了したとき，特別のキーシーケンスで実行を中断させたとき，またはTurbo Debuggerがブレークポイントに遭遇したときに，再びデバッガの制御に戻ります。

この章は，デバッガの制御のもとでプログラムの状態を調べる方法について説明します。またプログラムを部分的に実行するための方法と，プログラムの実行中に実行を中断する方法について説明します。最後に，同一のプログラムをデバッグするためにデバッグセッションを再開する方法と，別のプログラムをデバッグするためにデバッグセッションを開始する方法を説明します。

プログラムの現在の状態を調べる方法

プログラムの現在の状態は，次のような要素から構成されます。

■プログラムのDOSコマンドライン引数
■アクティブな関数または手続きのスタック
■ソースコードまたは機械語コードの中の現在の場所
■デバッガがプログラムの実行を停止した理由
■プログラムの変数の値

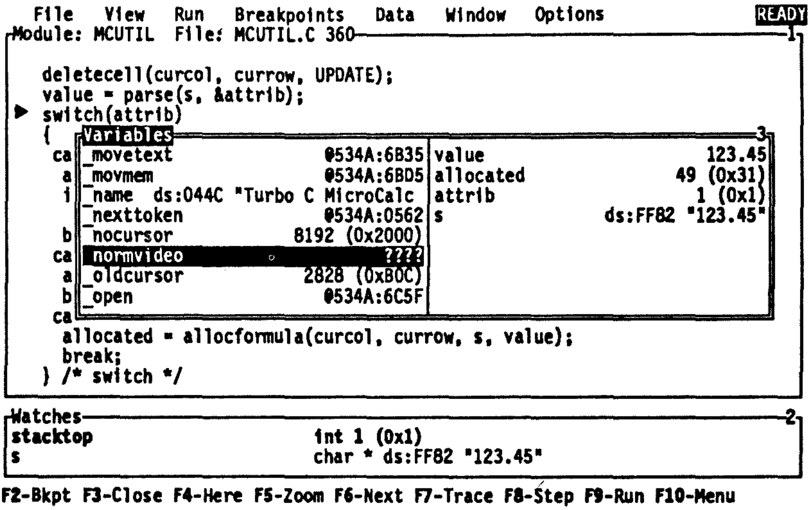
プログラムの変数の値を調べたり変更したりする方法については第6章を参照してください。以下の節では，Variablesウィンドウ，Stackウィンドウ，Globalペイン，Staticペインのローカルメニューコマンド，Originコマンド，Get infoコマンドについて説明します。

Variablesウィンドウ

このウィンドウは，プログラムのカレントロケーション（現在位置）からアクセスできるすべての変数（名前と値）を表示します。

名前が正確に思い出せない変数を探すときは，このウィンドウを使うと便利です。変数が見つかったら，ローカルメニューコマンドを使って，値を調べたり変更したりすることができます。またこのウィンドウを使って，呼び出された関数のローカル変数を調べることもできます。

図5.1　Variablesウィンドウ



注意：Turbo Pascalのプログラムをデバッグしているときは，右側のペインは，現在のコードセグメントのシンボルとIPレジスタ(*CS:IP*)を表示します。IPレジスタは，いつでもプログラムのカレントロケーションを指します。変数はアルファベット順に並べられません。

Variablesウィンドウは，メインメニューからView/Variableを選択して開きます。メインメニュー以外の場所にいるときは，***F10***を押してメインメニューに戻るか，***GRPH-V***を押してViewメニューに移り，ViewメニューからVariableを選択することができます。Variablesウィンドウには，2つのペインがあります。左側のペインは*Global*ペインで，プログラムのすべてのグローバル変数を表示します。右側のペインは*Static*ペインで，カレントモジュール（カレントロケーション(*CS:IP*)を含むモジュール）の中のすべての静的（static）変数を表示します。どちらのペインでも，左側に変数の名前を，右側に値を表示します。Turbo Debuggerがシンボルのデータ型に関する情報を見つけられないときは，図5.1に示すように，そのシンボルの値を表示する場所に4つの疑問符(????)を表示します。

Globalペインの中で***GRPH-F10***を押すと，Globalペインのポップアップメニューが現われます。コントロールキーによるショートカットが有効なときは，***CTRL***と希望するローカルメニューコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることができます。

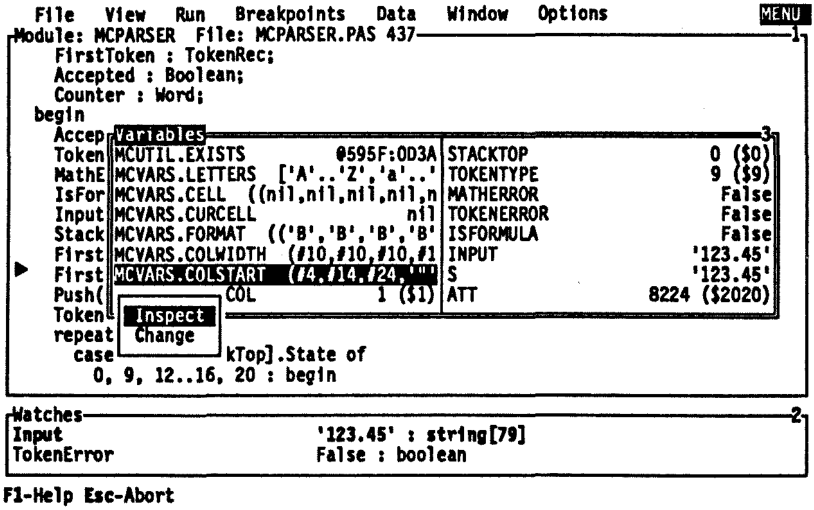
関数呼び出しで呼び出された関数にローカルな変数の値を見たい場合は，まずView/Stack

を選択してStackウィンドウを開き，その呼び出しにハイライトバーを移動します。次に***GRPH-F10***を押してローカルメニューを開きLocalsコマンドを選択します。するとVariablesウィンドウのStaticペインに，その呼び出しで呼び出されたときの，その関数のローカル変数の値が表示されます。

Globalペインのローカルメニュー

このローカルメニューには，InspectとChangeの2つのコマンドがあります。

図5.2　Globalペインのローカルメニュー



Inspect

Inspectorウィンドウを開き，Globalペインの中の現在ハイライト表示されているGlobalシンボルの内容を表示します。Inspectorウィンドウの詳しい説明は，第6章を参照してください。

ハイライト表示されている変数がルーチン名のときは，Inspectorウィンドウにその関数のソースコードを表示します。ソースファイルが見つからない場合は，CPUウィンドウに逆アセンブルされたコードを表示します。

ハイライト表示されている変数と同じ名前のローカル変数がある場合にも，ローカル変

数の値ではなく，グローバル変数の値を表示します。この動作は，Inspectorウィンドウの通常の動作と少し異なります。なぜならInspectorウィンドウは，通常カレントプログラムロケーションから見た変数の値を表示するからです。この機能のために，Inspectorウィンドウは，同じ名前がローカル変数としても使用されているグローバル変数の値を調べるための便利な方法になります。

Change

Globalペインで現在選択されている（ハイライト表示されている）グローバル変数の値を，入力された値に変更します。Turbo Debuggerは，必要に応じて，現在選択されている言語の代入演算子を使って代入したのとまったく同じように，データの型を変換します。代入とデータの型の変換の詳しい説明は，第9章を参照してください。

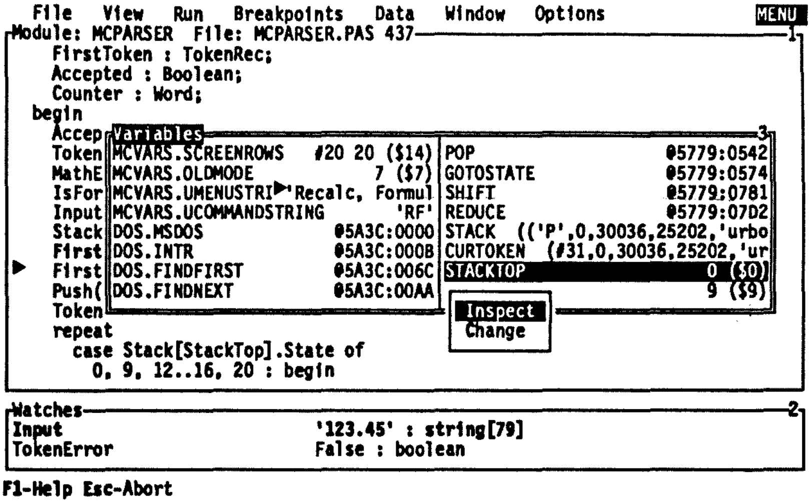
Changeコマンドを選択しないで，いきなり新しい値を入力することにより，現在ハイライト表示されているシンボルの値を変更することもできます。入力を始めると，Changeコマンドを選択したときと同じプロンプトボックスが現われます。

Staticペインのローカルメニュー

GRPH-F10を押すとStaticペインのローカルメニューが現われます。コントロールキーショートカットが有効なときは，***CTRL***キーと選択したいコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることができます。

Staticペインには，InspectとChangeの2つのローカルメニューコマンドがあります。

図5.3 Staticペインのローカルメニュー



Inspect

Inspectorウィンドウを開き，Staticペインで現在ハイライト表示されているモジュールのローカルシンボルの内容を表示します。Inspectorウィンドウの動作の詳しい説明は，第6章を参照してください。

Change

Staticペインで現在選択されている(ハイライト表示されている)ローカルシンボルの値を，あなたが入力する値に変更します。TurboDebuggerは，必要に応じて，現在選択されている言語の代入演算子を使って代入したのとまったく同じように，データの型を変換します。代入とデータの型の変換の詳しい説明は，第9章を参照してください。

Changeコマンドを選択しないで新しい値を入力することにより，現在ハイライト表示されているシンボルの値を変更することもできます。入力を始めると，Changeコマンドを選択したときと同じプロンプトボックスが現われます。

Stackウィンドウ

StackウィンドウはF10/View/Stackと選択することにより開きます。Stackウィンドウは，すべてのアクティブな関数や手続きのリストを表示します。このリストには，先頭に最後に呼び出されたルーチン，次にそのルーチンを呼び出したルーチン，その次に2番目のルーチンを呼び出したルーチンの順に，そのプログラムで最初に実行される関数または手続き(Pascalのプログラムではメインボディ，Cのプログラムでは普通**main**()と呼ばれる関数)までさかのぼって表示されます。各関数または手続きには，呼び出し引数の値が表示されます。

図5.4　Stackウィンドウ

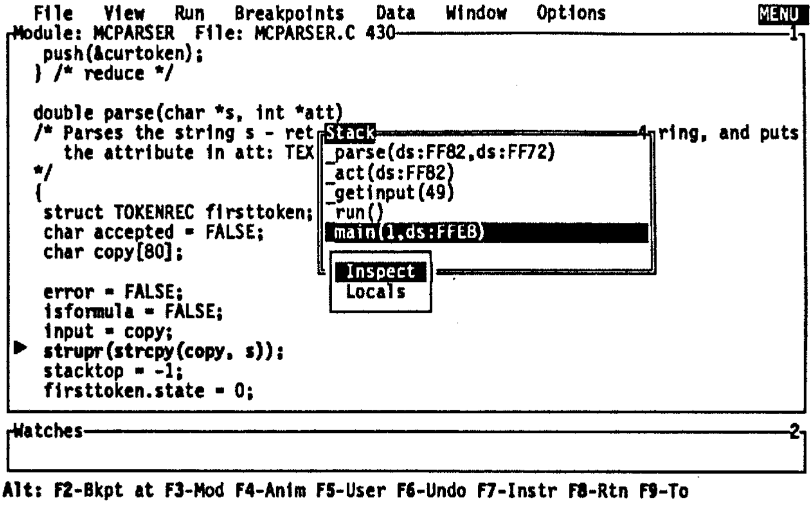
```
  File  View  Run  Breakpoints  Data  Window  Options                  READY
┌Module: MCPARSER  File: MCPARSER.C 430─────────────────────────────1┐
│  push(&curtoken);                                                   │
│ } /* reduce */                                                      │
│                                                                     │
│ double parse(char *s, int *att)                                     │
│ /* Parses the string s - ret┌Stack──────────────4┐ring, and puts    │
│    the attribute in att: TEX│ parse(ds:FF82,ds:FF72)│                │
│ */                          │ act(ds:FF82)        │                 │
│ {                           │ getinput(49)        │                 │
│  struct TOKENREC firsttoken;│ run()               │                 │
│  char accepted = FALSE;     │ main(1,ds:FFE8)     │                 │
│  char copy[80];             │                     │                 │
│                             └─────────────────────┘                 │
│  error = FALSE;                                                     │
│  isformula = FALSE;                                                 │
│  input = copy;                                                      │
│▶ strupr(strcpy(copy, s));                                           │
│  stacktop = -1;                                                     │
│  firsttoken.state = 0;                                              │
└─────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────2┐
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

GRPH-F10を押すとStackウィンドウのローカルメニューが現われます。また***CTRL***とコマンドの先頭文字を一緒に押すと，そのローカルメニューコマンドに直接アクセスすることができます。

Stackウィンドウのローカルメニュー

このメニューには，InspectとLocalsの2つのコマンドがあります。

図5.5 Stackウィンドウのローカルメニュー



Inspect

Moduleウィンドウを開き,現在ハイライト表示されている関数の中のアクティブラインを表示します。ハイライト表示されている関数がリストの最上位(最後に呼び出された)の関数の場合は,カレントプログラムロケーション(*CS:IP*)を表示します。ハイライト表示されている関数が最後に呼び出された関数でない場合は,その関数の,呼び出した関数から戻った後に実行される行を表示します。

ハイライトバーで関数を指定して**リターン**を押すことにより,このコマンドを呼び出すこともできます。

Locals

Variablesウィンドウを開いて,現在ハイライト表示されている関数のローカルシンボルと,カレントモジュールのローカルシンボルを表示します。

関数が自分自身を再帰的に呼び出している場合には,Stackウィンドウに,その関数が複数個連続して表示されます。ハイライトバーでそのうちの1つを指定してこのコマンドを使

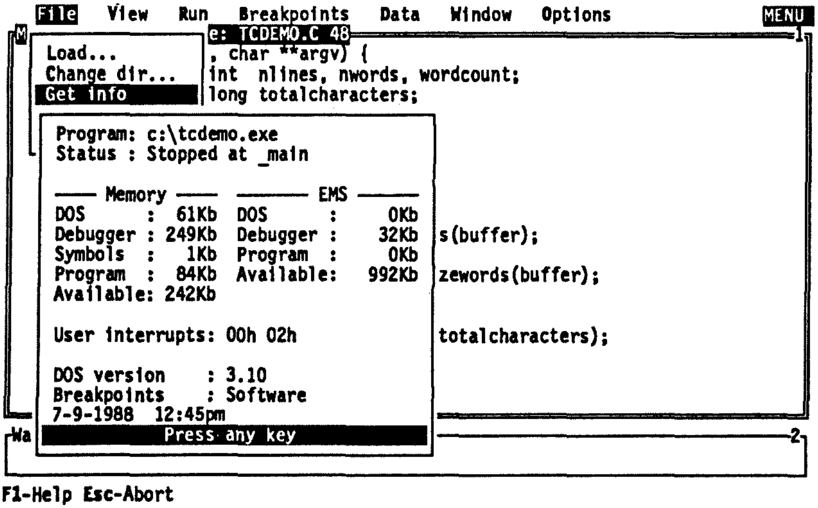
うと，その関数の，その呼び出しで呼び出されたときのローカル変数の値を見ることができます。

Originローカルメニューコマンド

Moduleウィンドウのローカルメニューと，CPUウィンドウのCodeペインのローカルメニューには，Originコマンドが入っています。Originコマンドは，カーソルをカレントプログラムロケーション(*CS:IP*)に移動します。このコマンドは，プログラムのコードをあちこち見たあと，プログラムが停止したロケーションに戻りたいときに非常に便利です。

Get infoコマンド

図5.6　File/Get infoコマンド



File/Get infoを選択することにより，メモリの使用状況を調べたり，デバッガが制御を取得した理由を調べたりすることができます。このコマンドにより表示される情報は次のとおりです。キー(どのキーでもよい)を押すと，表示は消えます。

- ■デバッグしているプログラムの名前
- ■プログラムの状態(プログラムが停止した理由)
- ■DOS, Turbo Debugger, プログラムが使用しているメモリの大きさ
- ■EMSメモリが増設されているときは，その使用状況(メインメモリの使用状況の右側に表示されます)
- ■デバッグしているプログラムの実行を停止させたすべての割り込みのリスト
- ■使用しているDOSのバージョン
- ■ブレークポイントが完全にソフトウェアにより処理されているか，またはハードウェアに支援されているか

2行目には，あなたのプログラムが停止した理由を示すメッセージが表示されます。表示されるメッセージは次のとおりです。

Stopped at

プログラムが，Run/Execute to，Run/Go to cursor，またはRun/Until returnコマンドの完了の結果として停止したことを示します。またプログラムがロードされ，ソースコードの先頭に到達するまでプログラムのコンパイラスタートアップコードが実行されたときにも，このメッセージが表示されます。

No program loaded

デバッグするプログラムを指定せずにTurbo Debuggerを起動したことを示します。プログラムをロードするか，CPUウィンドウのCodeペインのなかでAssembleローカルメニューコマンドを使って何か命令をアセンブルするまで，何も実行することはできません。

CTRL-STOP (IBM PCではCtrl-Break)

CTRL-STOP (***Ctrl-Break***) でプログラムの実行を停止させたことを示します。

Trace

F7または***F10***/Run/Traceで，ソースコードを1行または機械語命令を1命令実行したことを示します。

Breakpoint at __

実行を停止させるブレークポイントで実行が停止したことを示します。"at"の後ろには実行が停止したアドレスが表示されます。

Terminated, exitcode __

プログラムの実行が終了したことを示します。"code"の後ろにはプログラムがDOSに返す数値の終了コードが表示されます。プログラムが明示的に値を返さないときは，無意味な値が表示されることもあります。プログラムの実行が終了したときは，***F10***/Run/Program resetでプログラムを再ロードするまでは，プログラムを実行できません。

Loaded

Turbo Debuggerに，コンパイラスタートアップコードの実行を抑制するオプションを指定して，プログラムをロードしたことを示します。この時点では，プログラムのスタックやセグメントレジスタを設定する命令も含めて，命令はひとつも実行されていません。この段階でプログラムの中のデータを調べても，見ている値が正しいとは限りません。

Step

F8または***F10***/Run/Step overにより，関数呼び出しをスキップしながら，ソースコードを1行または機械語命令を1命令実行したことを示します。

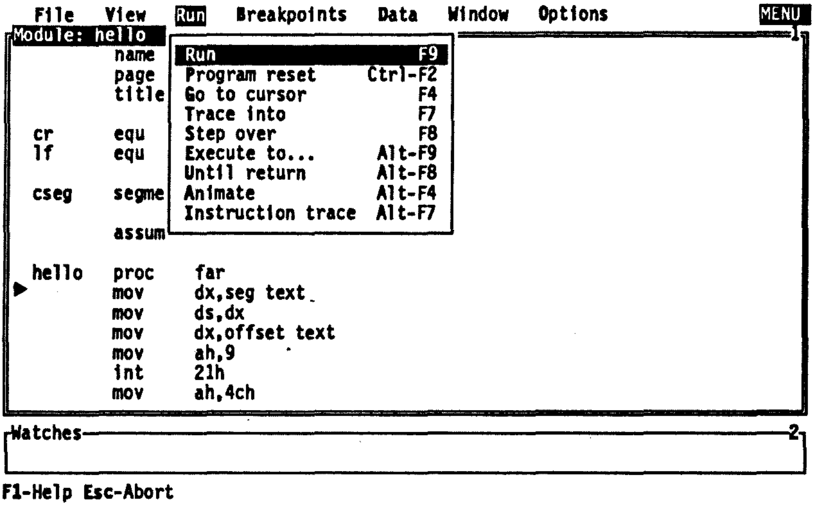
Interrupt

割り込みキー（通常は，PC-9801では***CTRL-STOP***，IBM PCでは***Ctrl-Break***）を押して実行を停止したことを示します。割り込みキーを押すと，プログラムの実行はただちに停止し，デバッガに制御が戻ります。

Runメニュー

Runメニューには，プログラムをいろいろな方法で実行するための多数のコマンドがあります。

図5.7　Runメニュー



Run　[F9]

プログラムの実行を開始します。次のイベントのどれかが発生したときは，デバッガに制御が戻ります。

■プログラムが終了したとき

■プログラムを停止させるブレークポイントに遭遇したとき

■*CTRL-STOP*が押されたとき

Program reset [CTRL-F2]

デバッグしているプログラムを，ディスクから再ロードします。プログラムを実行しすぎたとき，すなわちバグのある場所を通りすぎたときは，このコマンドを使って，最初からやり直すことができます。

Moduleウィンドウまたは CPUウィンドウにいるときは，表示はプログラムの先頭に戻りません。Program resetコマンドを選択したときにいた場所が表示されます。ソース文を1行余計に実行したためにProgram resetコマンドを選択した場合には，ソースファイルの中でカーソルを2,3行上に移動し，**F4**を押してその場所まで実行することができます。

Go to cursor [F4]

Moduleウィンドウの中またはCPUウィンドウのCodeペインの中のカーソルが置かれている行まで，プログラムを実行します。カレントウィンドウがModuleウィンドウの場合は，カーソルが，関数の内部のソースコード行に置かれていなければなりません。

Trace into [F7]

ソースコードを1行または機械語命令を1命令実行します。通常はソースコードを1行実行します。カレントラインが関数（または手続き）呼び出しを含むときは，Turbo Debuggerは，そのルーチンの中に入っていきます。ただしカレントウィンドウがCPUウィンドウの場合には，機械語命令を1命令だけ実行します。

Step over [F8]

関数（手続き）呼び出しを飛び越しながら，ソースコードを1行または機械語命令を1命令実行します。普通は，ソースコードを1行実行します。ただしカレントウィンドウがCPUウィンドウの場合には，機械語命令を1命令だけ実行します。

ソースコードの行を実行するときは，Turbo Debuggerは，その行の中の関数（手続き）呼び出しにより呼び出される関数または手続きをその行の一部として扱います。そのため，呼び出された関数や手続きの先頭では停止せず，カレントルーチンのその次の行，またはカレントルーチンを呼び出したルーチンの，カレントルーチンを呼び出した行の次の行で停止します。

機械語命令を実行するときは，いくつかの命令を，複数の命令の実行を引き起こす場合

にも，1命令として扱います。Turbo Debuggerが1命令として扱う命令は次のとおりです。

CALL	サブルーチンコール(near，far両方)
INT	インタラプトコール
LOOP	CXカウンタによるループ制御
LOOPZ	CXカウンタによるループ制御
LOOPNZ	CXカウンタによるループ制御

また**REP**，**REPZ**，**REPNZ**の後に**CMPS**，**CMPSB**，**CMPSW**，**LODSB**，**LODSW**，**MOVS**，**MOVSB**，**MOVSW**，**SCAS**，**SCASB**，**SCASW**，**STOS**，**STOSB**，**STOSW**が続くときもスキップします。

Execute to [GRPH-F9]

指定されたアドレスまでプログラムを実行します。ブレーク動作をするブレークポイントに先に遭遇した場合，および途中で実行を停止させた場合には，指定したアドレスまで実行されません。

Until return [GRPH-F8]

カレント関数から呼び出し側に戻るまで，プログラムを実行します。このコマンドは，***F10***/Run/Stepではなく***F10***/Run/Traceを選択して，意図しない関数や手続きの中に入り込んだときや，カレント関数が正しく動作することがわかり残りの部分を調べる必要がないときに便利です。

Animate [GRPH-F4]

Traceコマンドの動作を繰り返し，1回ごとに画面を更新します。このコマンドを選択すると，1つのTraceコマンドの実行が終了してから次のTraceコマンドの実行が開始されるまでの遅延時間の入力を求めます。遅延時間は，1/10秒単位で指定することができます。デフォルトは3です。

このコマンドを使うと，ソースコードのカレントロケーションを見ながら，変数の値が変化するのを監視することができます。キー(どのキーでもよい)を押すと，停止させることができます。

Instruction trace [GRPH-F7]

命令を1つだけ実行します。割り込みをトレースしたいときや，Moduleウィンドウの中にいて，モジュールの中の関数や手続き（たとえばライブラリルーチン）をデバッグ情報なしにトレースしたいときに，このコマンドを使ってください。

プログラムの実行を停止させる方法

会話型のプログラムでは，プログラムを実行し，コードの目的とする部分に到達するまでプログラムに応答し，その部分に到達したら実行を停止することにより，プログラムの目的とする場所に最も速く到達できる場合があります。この方法は，調べようとする部分が，着目している呼び出しの前に何回か呼び出されている場合にとりわけ有効です。

また，何か予想もしない理由によりデバッガに制御が戻らないため，プログラムの実行を停止させたいこともあります。コードのある部分に無限ループが存在するときは，このようなことが起り得ます。あるコードが実行されることを期待して，そこに実行を停止させるブレークポイントを設定してあっても，そのブレークポイントのところまで実行されないからです。

このような場合には，次の方法でプログラムの実行を停止させます。

CTRL-STOP（Ctrl-Break）

CTRL-STOPキー（IBM PCでは***Ctrl-Break***）により，ほとんどいつでもプログラムの実行を停止し，デバッガに制御を戻すことができます。このキーは押されると即座に機能するので，コードの意外な場所にいるように見えることがあります。たとえば，プログラムがキー入力を待っているときは，ROMのキーボードBIOSであったり，ループが実行されていたときはそのループの中のどれかの命令であったりします。

次の2つの状態にあるときは，***CTRL-STOP***でプログラムを停止させることはできません。***CTRL-STOP***でプログラムを停止させることができないときは，システムをリブートしてください。

■割り込みが禁止された状態で，プログラムがループにはまり込んでいる。

■エラーのあるコード(命令)が実行されたために，システムが壊れている。

CTRL-STOPでプログラムを停止させることができないときは，システムをリブートしてください。

CTRL-STOPを使用するプログラムをデバッグするときは，割り込みキーを変更する必要があります。TDINSTインストールプログラムを使うと，割り込みキーを変更することができます。割り込みキーは，***CTRL***と文字キーのどれかの組み合わせに割り当てることができます。

プログラムを終了させる

プログラムが終了してDOSに戻るときに，デバッガが再び制御を取り戻します。デバッガは，プログラムがDOSに返すメッセージを表示します。プログラムが終了すると，**Run/Program reset**でそのプログラムを再ロードするまでは，**Run**メニューのどのオプションも使うことができません。

セグメントレジスタとスタックは，プログラムが終了したあとは，たいてい正しくありません。ですから，プログラムが終了したあとは，プログラム変数を調べたり，変更したりしないでください。

デバッグセッションを再開する

プログラムをデバッグするときは，問題の原因となっている場所を通り過ぎて，少し先まで実行してしまうことがよくおこります。このような場合には，もう一度最初からやり直して，前回より少し前の行で実行を停止したい思われるでしょう。ほとんどのデバッガでは，デバッグセッションを再開して，前回と同じ場所またはその近くに到達するために，多数のコマンドをもう一度入力しなければなりません。それに対してTurbo Debuggerには，前回のセッションのキー入力を記録し，それを再生することができる強力な機能があります。

またTurbo Debuggerには，前回のプログラムを，前回と同じコマンドライン引数で再ロードする機能もあります。

プログラムを再ロードする

F10/Run/Program resetを選択すると，前回と同じプログラムを再ロードすることができます。Turbo Debuggerは，そのプログラムをディスクから再ロードします。前回のセッションで付け加えたデータがあるときは，そのデータはそのままの状態に保たれます。この方法は，デバッグを再開する最も安全な方法です。デバッグを再開してプログラムを先頭から実行すると危険な場合があります。なぜなら多くのプログラムは，データのうちあるものを初期化して，プログラムのディスクイメージから変更するからです。Program resetコマンドは，ブレークポイントおよびウオッチの設定をそのままに保ちます。

キーストロークの記録と再生

キーストロークマクロ機能を使うと，使用頻度の高い一連のキー入力を記録することができます。デバッグ中は，プログラムのある場所に行くために，一連のコマンドを繰り返し入力することがよくあります。これは非常にめんどうなものです。

めんどうなキー入力を繰り返さなくても済むように，デバッガを起動させてからプログラムが希望する状態になるまでのすべてのキー入力を，キーストロークマクロとして定義することができます。このようなキーストロークマクロを定義すると，それ以後プログラムの同じ場所にりたいときには，そのキーストロークマクロを再生するだけで済みます。

この機能は，ユーザプログラムに対して入力しなければならないキーストロークを記録するために使うことはできません。Turbo Debuggerのコマンドを入力するキーストロークにしか使えません。

デバッグセッションの全体を記録するには，DOSからTurbo Debuggerを起動したあと，まず**Options/Macros/Create**を選択します。するとプロンプトが，キーストロークマクロを割り当てるキーを押すよう求めます。まだ何も設定されていないキー，たとえば***SHIFT***とファンクションキー***F1***の組み合わせ***SHIFT-F1***を選んで，そのキーを押します。次にプログラムを，クラッシュが起きる場所の直前まで実行します。その場所まで到達したら，**Macros/Stop recording**を選択してキーストロークマクロの記録を停止します。最後に**Options/Save options**コマンドを選んで，このマクロをディスクに保存します。その後さらにプログラムの実行を続けます。プログラムがクラッシュしたら，Turbo Debuggerとプログラムを再ロードして ***SHIFT-F1***を押すと，記録したコマンド列によりプログラムが実行されます。

プログラムの実行を進めるためにキー入力が必要な場合には，その入力は手で行なわなければなりません。キー入力を必要としないプログラムは，前回実行された点まで自動的に実行され，デバッグの再開作業が完全に自動化されます。このように，このキーストローク記録機能を使うとキー入力数を大幅に減らすことができます。

注意：マクロの記録中にViewウィンドウを開き，そのウィンドウを開いたまますべてのオプションの設定をコンフィギュレーションファイルに記録すると，次にその設定が読み込まれたとき，マクロの実行によりViewウィンドウが開く前に自動的にViewウィンドウが開きます。

デバッグのために新しいプログラムをロードする

F10/File/Loadを選択すると，デバッグのために新しいプログラムをロードすることができます。DOS形式のワイルドカードを使ってファイルのリストを表示させ，その中から選択することもできますし，ロードするファイルの名前を入力することもできます。

プロンプトが現われてから**リターン**を押すと，カレントディレクトリの中のすべての.EXEファイルのリストが表示されます。ロードしたいファイルにハイライトバーを移動し**リターン**を押します。

ロードしたいファイルの名前を入力すると，入力した文字（何文字でもよい）で始まる名前のファイルに，ハイライトバーが移動します。

DOSのコマンドラインで指定するのとまったく同じようにプログラム名のあとに引数を入力することにより，プログラムに引数を指定することもできます。

```
myprog a b c
```

この例は，3つのコマンドライン引数*a*, *b*, *c*を指定してプログラム*MyProg*をロードします。

プログラムの引数を変更する

プログラムをロードするときにそのプログラムの必要な引数を指定するのを忘れたとき，または指定した引数を変更したいときは，**F10**/Options/Argumentsを使って引数を指定したり，変更したりすることができます。DOSのコマンドライン上で，プログラムの名前のあとに入力するのとまったく同じように，新しい引数を入力してください。

新しい引数を入力すると，Turbo Debuggerは，プログラムをディスクから再ロードするかどうか尋ねてきます。答えは普通Yesのはずです。なぜならほとんどのプログラムは，起動するときにだけコマンドライン引数を読み込むからです。

第6章

データを調べる，変更する

Turbo Debuggerは，ユーザプログラムのデータを詳しく調べるための，独自の視覚化された方法を提供します。Inspectorウィンドウでは，プログラムのデータを，ソースファイルに記述されているとおりに調べることができます。ポインタの指し示す先をたどったり，配列を端から端まで順に表示させたりすることができます。また構造体，レコード，共用体を，ソースファイルに書かれたとおりに見ることができます。また変数や式をWatchesウィンドウの中に置き，プログラムの実行が進行するにつれてそれらの値が変化するのを監視することができます。

この章は，使用している言語(C，Pascal，またはアセンブラ)のいろいろなデータ型についての十分な理解を前提としています。しかし，その言語を使い始めたばかりで，まだその言語のすべてのデータ型をよく知らない場合にも，基本的なデータ型（char，int，integer，boolean，realなど)について貴重な情報を得ることができるでしょう。複雑なデータ型についてもよくわかるようになったら，あらためてこの章に戻り，Turbo Debuggerでそれらのデータを見る方法について習得してください。

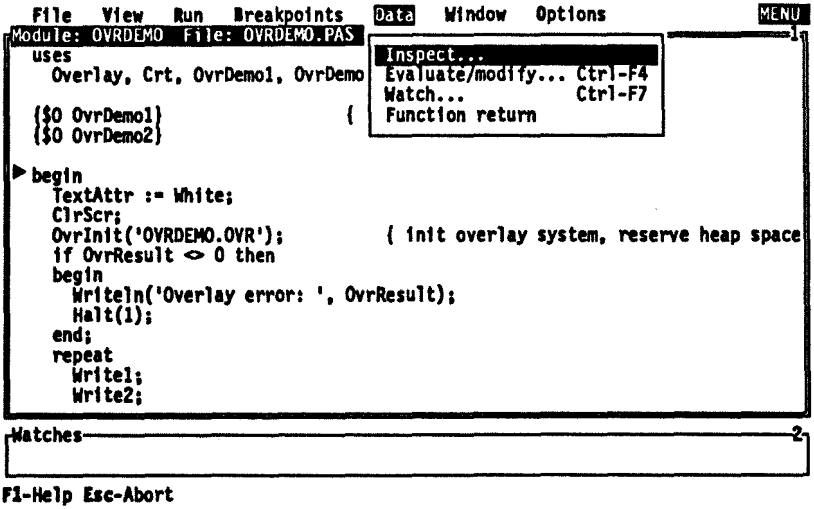
この章では，ユーザプログラムの変数を調べたり，値を変更する方法を説明します。まず，Dataコマンドとそのオプションについて説明します。それから，副作用をともなう式を評価することにより，プログラムデータを変更する方法を説明します。次に，ソースモジールの中のデータに，直接アクセスする方法を説明します。さらにWatchesウィンドウについて説明します。最後に，各言語の基本的なデータ型が，Inspectorsウィンドウでどのように見えるかを示します。

メモリ上のあるブロックを16進数データバイトとして調べたり，変更したい場合は，アセンブラレベルのデバッグについて説明している第10章を参照してください。

Dataメニュー

Dataメニューには，プログラムのデータを調べたり，変更したりするためのコマンドが入っています。このメニューにより，式を評価したり，変数の値を変更したり，Inspectorウィンドウを開いてデータの内容を表示させたりすることができます。

図6.1　Dataメニュー



Inspect

調べたい変数を入力するように求め，入力されたらInspectorウィンドウを開いて，その変数の内容を表示します。変数名だけでなく，複雑な式も入力することができます。ただし式は，評価の結果がただ定数になるのではなく，メモリの場所を指さなければなりません。

このコマンドを選択したときにカーソルがTextペインにあると，カーソルの置かれている変数(もしあれば)が，自動的にプロンプトのところに表示されます。また***INS***を使って式を指定したときは，その式がプロンプトのところに現われます。

構造体の配列や共用体のような複雑なデータ構造を調べるときは，そのデータ構造を調べるのに適したInspectorウィンドウが現われます。そしてそのInspectorウィンドウの中で，そのデータの構成要素をさらに詳しく調べることができます。そのため，Moduleウィンドウのソースコードで見るのと同じくらい容易に，データの構造を追って行くことができます。

Inspectorウィンドウの働きについては，この章の後半にある「**Inspectorウィンドウ**」の節を参照してください。

Evaluate/modify

評価する式を入力するように求めます。式が入力されると，Turbo Debuggerはコンパイラと同じ方法で式を評価します。式についての詳しい説明は第9章を参照してください。

このコマンドを選択したときにカーソルがTextペインにあると，カーソルの置かれている変数が（もしあれば），自動的にプロンプトのところに表示されます。また***INS***を使って式を指定したときは，その式がプロンプトのところに現われます。

ウォッチしようとする式には，その後に書式文字列を付け加えることができます。書式制御については第9章を参照してください。ウォッチしたいデータを，そのデータ型に対してあらかじめ決められている書式とは異なる書式で表示させたいときに便利です。

プロンプトボックスには，3つのペインがあります。一番上のペインに，評価したい式を入力します。このペインは，他の入力ペインと同じように，ヒストリリストを持っています。中間のペインには，入力した式を評価した結果が表示されます。一番下のペインは，式の変数に新しい値を入力することができる入力ペインです。式が変更できないものであるときは，Cannot be changed（変更できない）というメッセージが表示され，このペインにカーソルを移動することはできません。

複数のペインを持つ他のウィンドウにいるときと同じように，***TAB***と***SHIFT-TAB***を使って，ペインからペインに移ることができます。EvaluateペインおよびNew Entryペインへの入力は，**リターン**を押したときに有効になります。どのペインにいるときでも，***ESC***を押すと，プロンプトボックスが閉じます。

結果は，そのデータ型に適した書式表示されます。結果を別の書式で表示させるには，式の後に区別文字(,)を入力し，その後に書式文字列を入力してください。書式文字列の詳しい説明は，第9章を参照してください。

プログラム変数の名前を単独で入力したときは，その変数の現在の値が表示されます。変数の値を素早く見たいときは，Inspectorウィンドウを開いて，値を見て，閉じるよりも，この方法のほうが便利です。またこのコマンドは，オペランドとしてプログラム変数ではなく数を入力することにより，簡単な電卓として使うこともできます。

Cプログラマへの注意

C言語には，副作用（*side effects*）のある式と呼ばれる機能があります。この機能は，驚きと混乱の原因にもなり得るものですが，強力で便利な機能でもあります。

副作用のある式は，評価されたときに，1つまたは複数の変数値すなわちメモリ領域の内容を変化させます。たとえば，Cのインクリメント演算子(＋＋)，デクリメント演算子(--)，代入演算子(=，＋=，など)が，この効果を引き起こします。Cの式中でプログラムの中の関数(たとえば**myfunc(2)**)を実行する場合には，その関数が予期しない副作用を引き起こす可能性があることに注意してください。

プログラム変数を含む式を評価したいだけで，変数の値を変更するつもりがない場合には，副作用のある演算子は使わないでください。しかし一方でこの副作用は，変数の値(メモリ領域の内容)を変更する手軽な方法でもあります。たとえば，Cの式***count***＋＋を評価すると，変数***count***の値に1を加えることができます。

Watch

監視する式を入力するように求めます。入力されたら，その式または変数をWatchesウィンドウの中に表示されている変数のリストに加えます。

このコマンドを選択したときにカーソルがTextペインにあると，カーソルの置かれている変数(もしあれば)が，自動的にプロンプトのところに表示されます。また***INS***を使って式を指定したときは，その式がプロンプトのところに現われます。

Function return

現在の関数が返す値を表示します。このコマンドは，現在の関数が呼び出し側に戻る直前でなければ使用できません。

戻り値はInspectorウィンドウ内に表示されます。したがって戻り値が複雑な構造のデータを指すポインタである場合に，ポインタの指すデータを容易に調べることができます。またCPUレジスタの中に置かれる戻り値を調べるために，CPUウィンドウに切り替える必要がなくなります。

ソースファイルの中のデータを指定する

Turbo Debuggerは，調べたいプログラム変数の名前をいつでも入力しなくても済むようにする，強力な機能を備えています。どのModuleウィンドウの中でも，変数名のどこかにカーソルを置き，ローカルメニューのInspectコマンドを選択することにより，その変数の内容を表示するInspectorウィンドウを開くことができます。また，Inspectコマンドを選択する前に，矢印キーと***INS***キーを使って調べたい式を指定することもできます。指定した式はハイライト表示されます。Moduleウィンドウの詳しい説明は第8章を参照してください。

Watchesウィンドウ

Watchesウィンドウには，値の変化を追跡するために指定された変数や式が表示されます。intのように単純な変数の値だけでなく，配列のように複雑な変数まで，ウォッチすることができます。さらに$x*y+4$のような計算式の値もウォッチできます。

図6.2　Watchesウィンドウ

```
  File   View   Run   Breakpoints   Data   Window   Options                   READY
Module: TCDEMO  File: TCDEMO.C 35──────────────────────────────────────────────1
          unsigned long totalcharacters;

          nlines = 0;
          nwords = 0;
          totalcharacters = 0;
          showargs(argc, argv);
          while (readaline() != 0) {
                  wordcount = makeintowords(buffer);
                  nwords += wordcount;
                  totalcharacters += analyzewords(buffer);
                  nlines++;
          }
 ►        printstatistics(nlines, nwords, totalcharacters);
          return(0);

Watches────────────────────────────────────────────────────────────────────────2
wordcount                      unsigned int 8 (0x8)
wordcounts                     unsigned int [10] {1,2,4,6,1,1,2,0,0,0}
letterinfo  struct linfo [26]  {{4,2},{1,1},{0,0},{1,1},{7,0},{2,2},{2,0},{5,0}
nlines*nwords                  unsigned int 34 (0x22)
totalcharacters                unsigned long 66L (0x42)
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

View/Watchesを選択すると，Watchesウィンドウが開きます。Watchesウィンドウには，値をウォッチしたい変数や式が表示されます。変数や式を左側に，そのデータ型と値を右側に表示します。配列や構造体の値のような複合値は，Cのプログラムの場合には中カッコ｛ ｝で囲んで，Pascalのプログラムの場合には丸カッコ( )で囲んで表示します。スペースが不足して変数名や式の全体を表示することができないときは，表示できない部分は省略し，1個の中点(·)により省略された部分があることを示します。

ウォッチしたい式を入力するときは，呼び出されていない関数の中にあるためまだ有効でない変数の名前も，自由に使うことができます。そのため，ウォッチしようとする式を，スコープ（通用範囲）がアクティブになる前に，設定することができます（変数やパラメータが有効になる時点とそのスコープについての詳しい説明は，第9章を参照してください）。Turbo Debuggerでは，すぐには評価できない式を入力することができるのはこの場

合だけです。

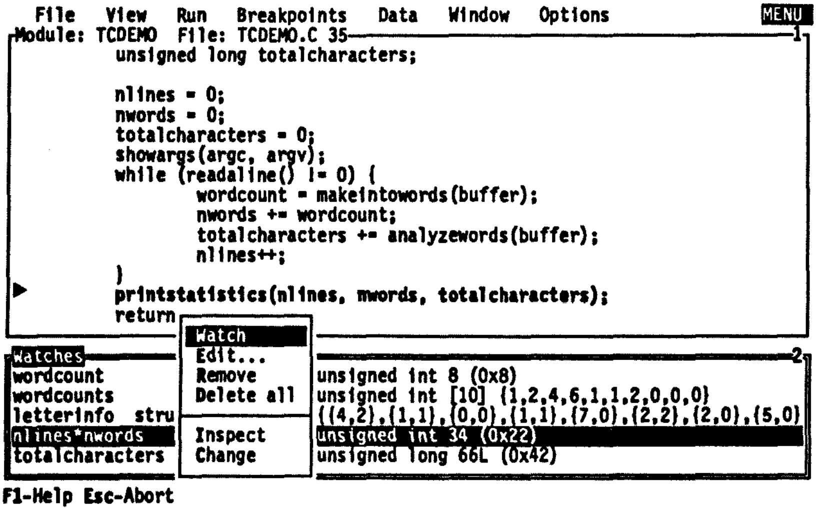
このことはまた，変数の名前の入力を間違えても，その間違いは検出されないことを意味します。なぜならTurbo Debuggerは，その変数名を，プログラムの実行が進むと有効になるかもしれない変数の名前であると見なすからです。

第9章に説明するスコープ(通用範囲)オーバーライド機構を使わないかぎり，Turbo Debuggerは，プログラムが停止されたカレントロケーションのスコープを適用して，Watchesウィンドウの中の式を評価します。ウォッチしている式が，現在のスコープの中ではアクセスできない変数（たとえば別のモジュール内部変数）を含む場合には，式の値は評価されず，4つの疑問符（????）が表示されます。

Watchesウィンドウのローカルメニュー

すべてのローカルメニューと同じように，***GRPH-F10***を押すとWatchesウィンドウのローカルメニューが表示されます。***CTRL***キーによるショートカットが有効なときは，***CTRL***キーとコマンドの先頭文字を一緒に押すと，そのコマンドにアクセスすることができます。

図6.3　Watchesウィンドウのローカルメニュー



Watch

Watchesウィンドウのリストに付け加える変数名または式を入力するように求めてきます。入力した変数名または式は，リストの先頭に付け加えられます。

Edit

Watchesウィンドウの中の項目の編集が可能になります。既存の項目を変更することもできますし，新しい変数や式を入力することもできます。

変更したい項目にハイライトバーを移動し**リターン**を押すことにより，このコマンドを実行することもできます。

Remove

Watchesウィンドウから現在選択されている項目を削除します。

Delete all

Watchesウィンドウからすべての項目を削除します。新たな変数や式をウォッチしたいときは，Watchコマンドを使って設定してください。このコマンドは，プログラムのある部分から別の部分に移行し，それまでウォッチしていた変数が不要になったときに便利です。

Inspect

Inspectorウィンドウを開いて，Watchesウィンドウの中の現在ハイライト表示されている項目の内容を表示します。その変数が複合データ（配列，レコード，構造体）のときは，このコマンドにより，Watchesウィンドウに表示されている要素だけでなく，すべての要素を見ることができます（この後の「**Inspectorウィンドウ**」の節にInspectorウィンドウの詳しい説明があります）。

Change

Watchesウィンドウの中の現在ハイライト表示されている項目の値を，プロンプトにしたがって入力された値に変更します。Turbo Debuggerは，必要に応じて使用している言語の代入演算子(=または：=)とまったく同じ方法でデータの型をキャスト（変換）します。代入演算子とキャストについての詳しい説明は，第9章を参照してください。

Inspectorウィンドウ

Inspectorウィンドウは，あなたが調べようとしているプログラムのデータを，そのデータの型に適した書式で表示します。スカラー（**char**，**int**など），ポインタ，配列，構造体，レコード，共用体，集合，関数などを，ソースファイルに書かれているとおりに見ることができます。

Inspectorウィンドウは，調べているデータの構成要素を一覧表示します。ウィンドウのタイトルは，データの名前(もしあれば)を表示します。

Inspectorウィンドウに表示される最初の項目は，その変数がレジスタに置くように最適化されていなければ，その変数のアドレスです。アドレスは，セグメント：オフセットの書式で表示されます。

Inspectorウィンドウの内容をそのままデータバイトとして調べるには，Inspectorウィンドウの中からView/CPUコマンドを選択してください。そうするとCPUウィンドウが開き，そのDataペインに，Inspectorウィンドウに表示されているデータが表示されます。Window/Closeコマンド（または***F3***）でCPUウィンドウを閉じると，Inspectorウィンドウに戻ります。

以下の節では，Turbo Debuggerがサポートする言語（C，Pascal，アセンブラ）それぞれにおけるInspectorウィンドウについて説明します。Inspectorウィンドウでは，表示されるデータの書式が，使用している言語によって決まります。データ名とその値が，ソースファイルで宣言されているのと同じ書式で表示されます。

Inspectorウィンドウの型を指定する必要はありません。使用している言語によって，自動的に正しいウィンドウが選択されます。

CのデータのInspectorウィンドウ

スカラー

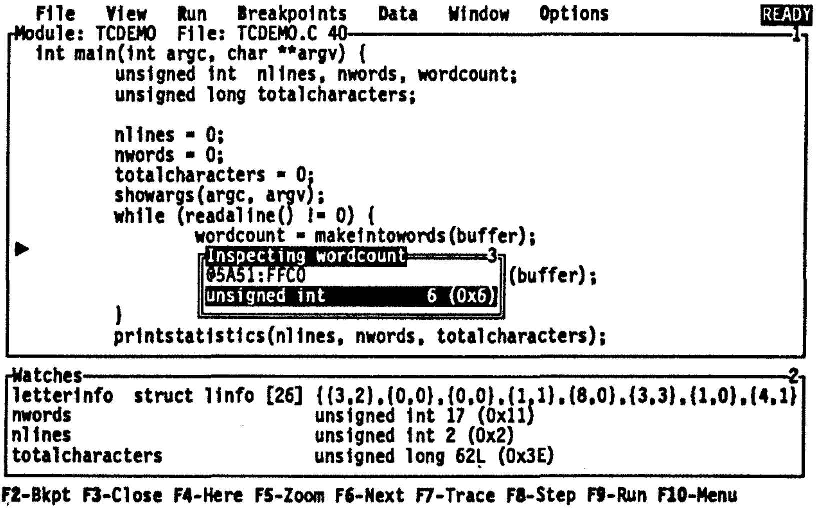
スカラーのInspectorウィンドウは，単純なデータ型の変数の値をたとえば次のように表示します。

```
char x = 4;
unsigned long y = 123456L;
```

このInspectorウィンドウには，変数のアドレスを表示する行の次に，ただ1行の情報しか表示されません。その行の左側には，そのスカラー変数の型（**char**, **unsigned long**など）が表示されます。また右側には，その変数の現在の値が表示されます。値は10進数，16進数，またはその両方で表示させることができます。普通は，まず10進数で表示され，その後に16進数で（Cの標準的な16進数の頭文字0xを使って）表示されます。TDINSTを使うと，値の表示書式を変更することができます。

表示されている変数が**char**型のときは，文字値（その値をコードとする文字）も表示されます。その変数の現在の値が文字として表示できないときは，円記号￥(IBM PCではバックスラッシュ＼）とそれに続く16進数を使って文字値を表示します。文字値は，10進数または16進数の前に表示されます。

図6.4　CのスカラーのInspectorウィンドウ



ポインタ

ポインタのInspectorウィンドウは，ポインタとポインタの指し示すデータに関する情報

を表示します。

```
char *p = "abc";
int *ip = 0;
int **ipp = &ip;
```

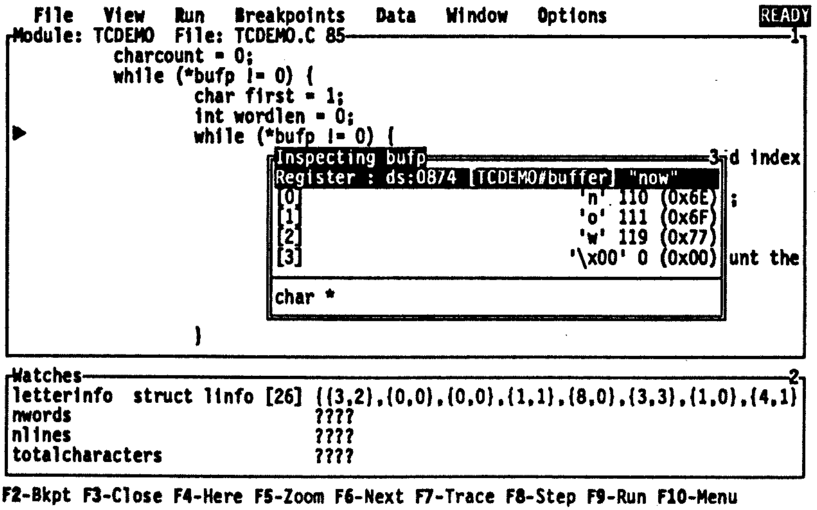
このInspectorウィンドウには，たいていの場合，変数のアドレスを示す最初の行の下にただ1行の情報しか表示されません。

この行の左側には，配列の最初の要素であることを示す［0］が表示されます。右側には，指し示されるデータの値が表示されます。

ポインタが指し示すデータが，配列や構造体のような複合型のときは，できるだけ多くの要素の値が，中カッコ｛ ｝で囲んで表示されます。

またポインタが**char**型で，ヌル文字で終了する文字列を指し示すときは，その文字配列の2番目以降の項の値も表示されます。各行の左側には，配列のインデックス（添字）が表示されます（［1］，［2］など）。右側には，そのインデックスにより指定される項の値が，スカラーのInspectorウィンドウと同じ書式で表示されます。またウィンドウの一番上の行には，変数のアドレスとその変数が指し示す文字列のアドレスとともに，文字列の全体が表示されます。

図6.5　CのポインタのInspectorウィンドウ



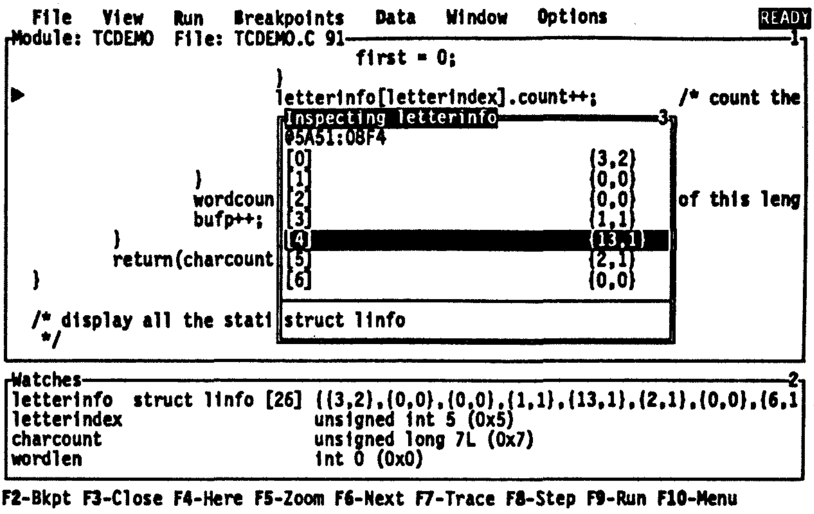
配列

配列のInspectorウィンドウは，配列型の変数の値を表示します。

```
long thread  [3] [4] [5];
char message [] = "eat these words";
```

配列の要素が各行に1つずつ表示されます。各行の左側には配列のインデックス（添字）が表示され，右側にはその索引で指定される要素の現在の値が表示されます。その値がまた構造体や配列のような複雑なデータ型のときは，そのレコードまたは配列のできるだけ多くの要素の値が，中カッコ｛ ｝で囲んで表示されます。

図6.6　Cの配列のInspectorウィンドウ



構造体と共用体

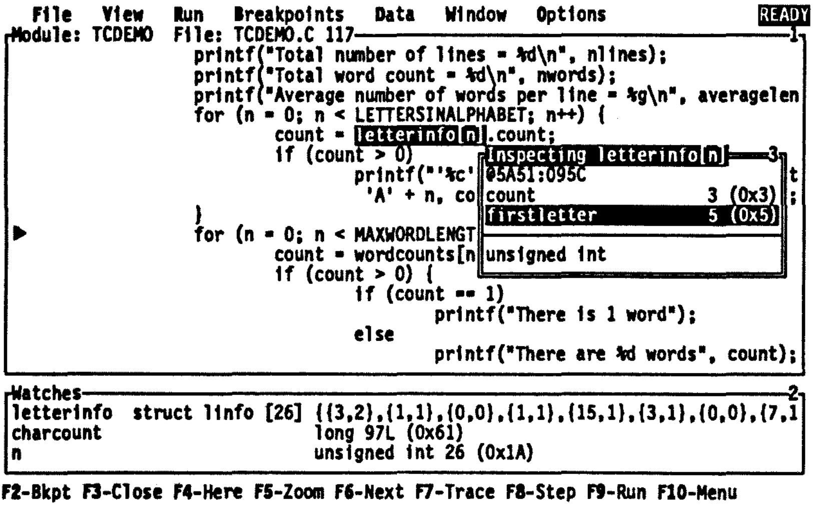
構造体および共用体のInspectorウィンドウは，構造体や共用体の要素の値を表示します。

```
struct date {
    int year;
    char month;
    char day;
}

union {
    int small;
    long gate:
} holder;
```

これらのInspectorウィンドウには，要素の値を示すペインの下に，もうひとつのペインがあります。このペインには，上のペインの中でハイライト表示されている要素のデータ型が表示されます。

図6.7　構造体および共用体のInspectorウィンドウ



構造体と共用体は，Inspectorウィンドウの中に同じように表示されます。Inspectorウィンドウのタイトルは，現在表示されているのが構造体であるか共用体であるかを示します。またアドレスが表示される行の下に，その構造体または共用体の要素と値が表示されます。左側に要素の名前が，右側にその値が，その要素のデータ型に適した書式で表示されます。

関数

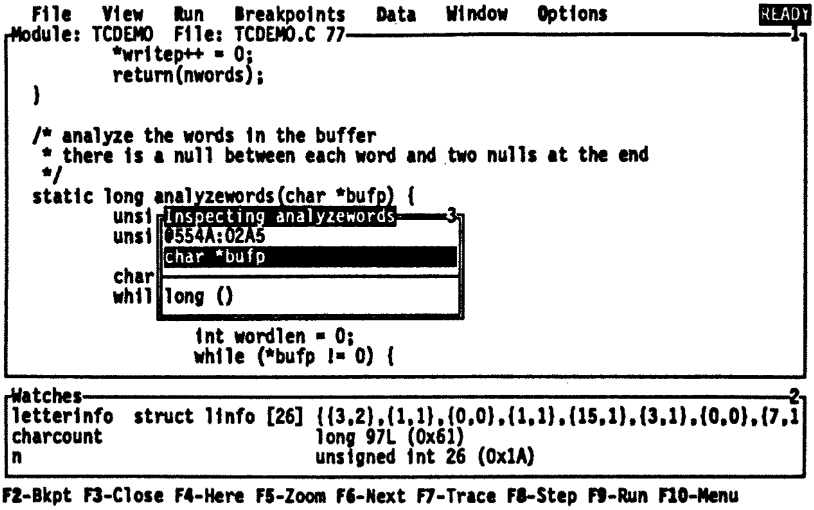
関数のInspectorウィンドウは，関数の呼び出しパラメータ，戻り値のデータ型，呼び出し方式に関する情報を表示します。

```
static int near pascal readit (char *buf, int count) {
}

char *nexterror (int errnum) {
}
```

タイトルの一部に，表示されている関数からの戻り値のデータ型が表示されます。またメモリアドレスの表示行の下に，その関数が呼び出されたときのパラメータが表示されます。

図6.8　Cの関数のInspectorウィンドウ



PascalのデータのInspectorウィンドウ

スカラー

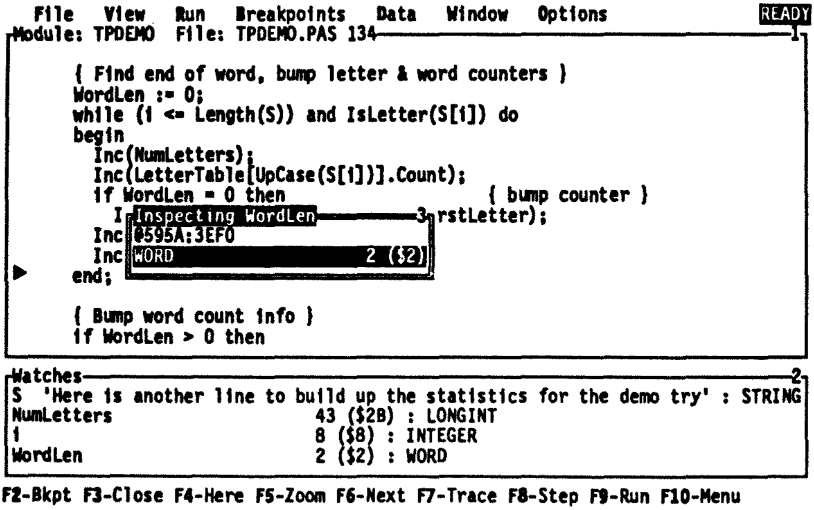
PascalのスカラーのInspectorウィンドウは，単純なデータ型の変数の値を表示します。

```
var
  X : integer;
  Y : longint;
```

このInspectorウィンドウには，変数のアドレスを表示する最初の行の次に，ただ1行の情報しか表示されません。その行の左側には，そのスカラー変数の型(**byte,word,integer, longint**など)が表示されます。また右側には，その変数の現在の値が表示されます。値は10進数，16進数，またはその両方で表示させることができます。普通は，まず10進数で表示され，その後に16進数で（**Turbo Pascal**の16進数の接頭辞＄を使って）表示されます。**TDINST**を使うと，値の表示書式を変更することができます。

表示されている変数が**byte**または**shortint**型のときは，数値とともに文字値（その値をコードとする文字）も表示されます。その変数の現在の値が文字として表示できないときは，シャープ記号＃とそれに続く16進数を使って文字値を表示します。文字値は，10進数または16進数の前に表示されます。

図6.9　PascalのスカラーのInspectorウィンドウ



ポインタ

PascalのポインタのInspectorウィンドウは，ポインタとポインタが指し示すデータに関する情報を表示します。

```
var
  IP : ^integer;
  LP : ^^pointer;
```

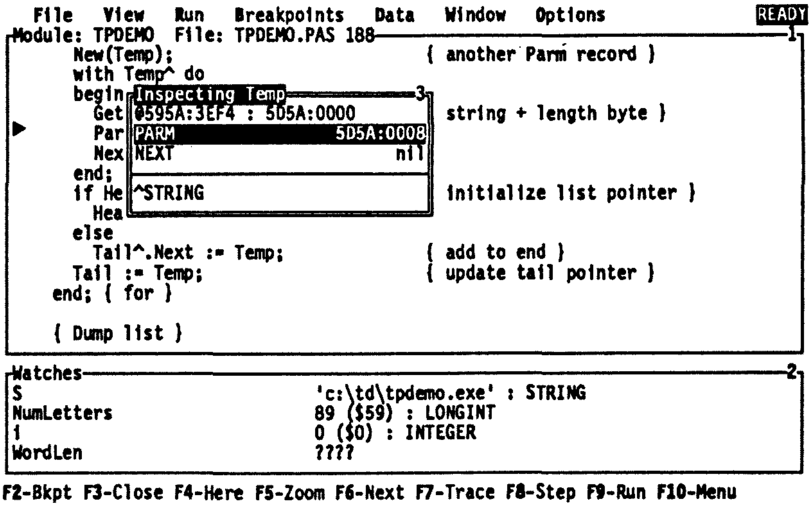
このInspectorウィンドウには，たいていの場合，変数のアドレスを示す最初の行の下に，ただ1行の情報しか表示されません。

この行の左側には，配列の最初の要素であることを示す［1］が表示されます。右側には，指し示されるデータの値が表示されます。

ポインタが指し示すデータが，レコードや配列のような複雑なデータ型のときは，できるだけ多くの要素の値が，丸カッコ( )で囲んで表示されます。

Inspectorウィンドウを開き，Rangeローカルコマンドを選択して，*count*に1よりも大きな値を指定したときも複数の行が表示されます。

図6.10 PascalのポインタのInspectorウィンドウ



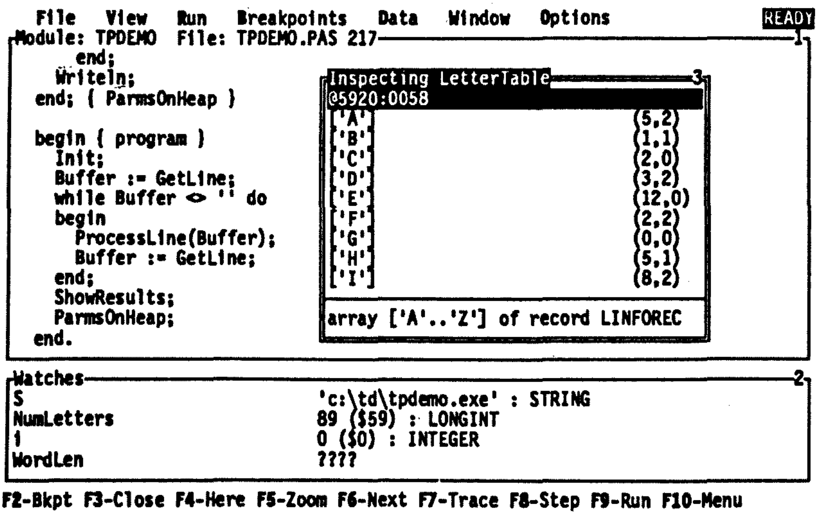
配列

Pascalの配列のInspectorウィンドウは，配列型の変数の値を表示します。

```
var
  A : array[1..10, 1..20] of integer;
  B : array[1..50] of boolean;
```

配列の要素が各行に1つずつ表示されます。各行の左側には配列のインデックス（添字）が表示され，右側にはそのインデックスで指定される要素の現在の値が表示されます。その値がまたレコードや配列のような複雑なデータ型のときは，そのレコードまたは配列のできるだけ多くのフィールドまたは要素の値が，丸カッコ( )で囲んで表示されます。

図6.11　Pascalの配列のInspectorウィンドウ



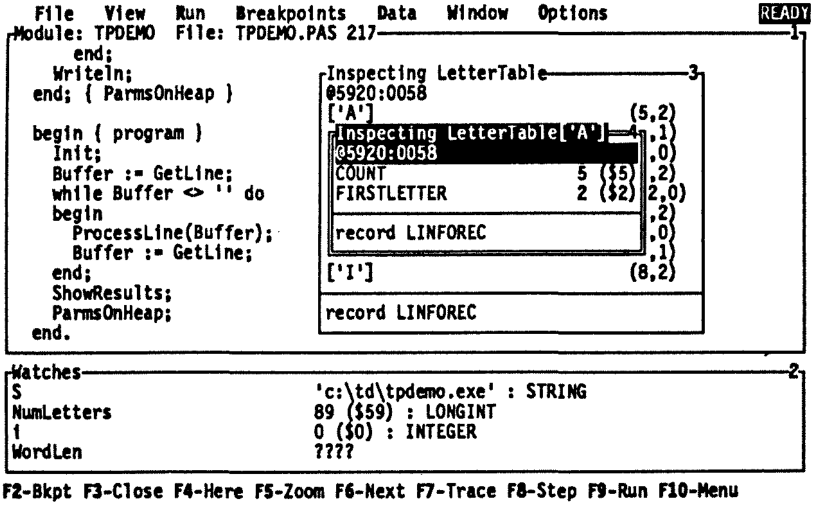
レコード

Pascalのレコードのインスペクタウィンドウは，レコードのフィールドの値を表示します。

```
record
  year : integer;
  month : 1..12;
  day : 1..31;
end
```

このInspectorウィンドウには，フィールドの値を示すペインのに下に，もうひとつのペインがあります。このペインには，上のペインの中でハイライト表示されているフィールドのデータ型が表示されます。

図6.12　PascalのレコードのInspectorウィンドウ



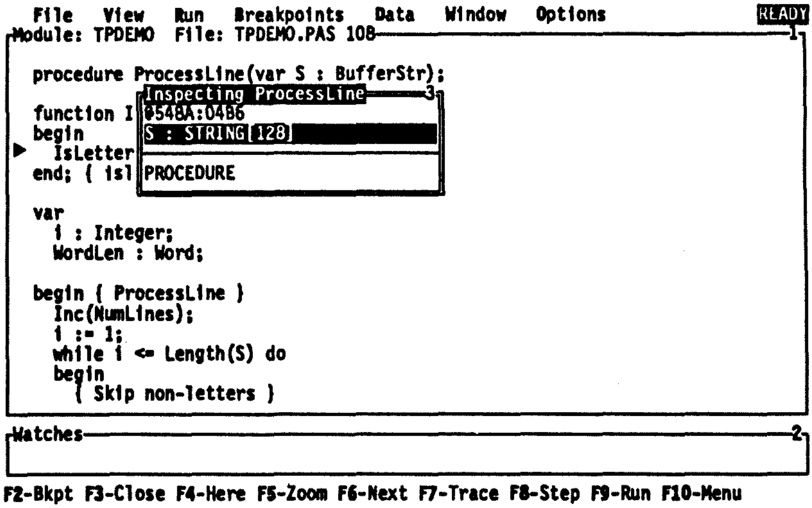
手続きと関数

Pascalの手続きおよび関数のInspectorウィンドウは，手続きまたは関数の呼び出しパラメータと戻り値のデータ型に関する情報を表示します。

```
function times2plus(a: integer, b: longint): longint
begin
  times2plus := a * 2 + b;
end;

procedure swap(var a,b : integer);
begin
  temp := a;
  a := b;
  b := a;
end;
```

図6.13　Pascalの手続きのInspectorウィンドウ



アセンブラのデータのInspectorウィンドウ

スカラー

アセンブリ言語のスカラーのInspectorウィンドウは，単純な変数の値を表示します。

```
VARI     DW 99
MAGIC    DT 4.608
BIGNUM   DD 123456
```

このInspectorウィンドウには，変数のアドレスが表示される最初の行の下に，ただ1行の情報しか表示されません。この行の左側には，そのスカラー変数の型（byte，word，dword，qwordなど）が表示されます。また右側には，その変数の現在の値が表示されます。値は10進数，16進数，またはその両方で表示させることができます。普通は，まず10進数で表示され，その後に16進数で（アセンブラの標準的な16進数接尾文字'h'を使って）表示されます。TDINSTを使うと，値の表示書式を変更することができます。

図6.14　アセンブラのスカラーのInspectorウィンドウ

```
  File   View   Run   Breakpoints   Data   Window   Options              READY
┌Module: hello  File: hello.asm 31──────────────────────────────────────────1┐
│           mov     ah,4ch                                                   │
│           mov     al,00h                                                   │
│           int     21h                                                      │
│ hello     endp                                                             │
│                                                                            │
│ cseg      ends                                                             │
│                                                                            │
│                                                                            │
│ dseg      segment para 'DATA'                                              │
│                                                                            │
│ text      db      'Hello World',cr,lf                                      │
│           db      '$'                                                      │
│ textptr   dw      text                                                     │
│ count     dd      12h                                                      │
│ ┌Inspecting count────────3┐                                                │
│ │@53CC:0010               │                                                │
│ │dword           18L (12h)│e'                                              │
│ └─────────────────────────┘                                                │
└────────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────2┐
│                                                                            │
└────────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

ポインタ

アセンブラプログラムのポインタのInspectorウィンドウは，他のデータを指し示すデータとそのデータが指し示すデータに関する情報を表示します。

```
X         DW    0
XPTR      DW    X
FARPRT    DD    X
```

このInspectorウィンドウには，たいていの場合，変数のアドレスを示す最初の行の下に，ただ1行の情報しか表示されません。

この行の左側には，配列の最初の要素であることを示す［0］が表示されます。右側には，指し示されるデータの値が表示されます。

ポインタが指し示すデータが，レコードや配列のような複雑なデータ型のときは，できるだけ多くの要素の値を中カッコ｛｝で囲んで表示しれます。

ポインタがbyte型で，ヌル文字で終了する文字列を指し示すときは，さらに文字配列の2番目以降の項の値も表示されます。各行の左側には，配列のインデックス（添字）が表示されます（[1]，[2] など）。右側には，その索引により指定される項の値が，スカラーのInspectorウィンドウと同じ書式で表示されます。またウィンドウの一番上の行には，変数のアドレスとその変数が指し示す文字列のアドレスとともに，文字列の全体が表示されます。

Inspectorウィンドウを開き，Rangeローカルコマンドを選択して，カウントに1よりも大きな値を指定したときも複数の行が表示されます。

図6.15　アセンブラのポインタのInspectorウィンドウ

```
  File  View  Run  Breakpoints  Data  Window  Options                  READY
┌Module: hello  File: hello.asm 14──────────────────────────────────────────1┐
│                                       ┌Inspecting textptr════════════3┐   │
│ hello   proc    far                   │@53CC:C00E : ds:0000 [hello.text]│  │
│         mov     dx,seg text           │[0]                  'H' 72 (48h)│  │
│         mov     ds,dx                 │[1]                 'e' 101 (65h)│  │
│         mov     dx,offset text        │[2]                 'l' 108 (6Ch)│  │
│►        mov     ah,9                  │[3]                 'l' 108 (6Ch)│  │
│         int     21h                   │[4]                 'o' 111 (6Fh)│  │
│         mov     ah,4ch                │[5]                 ' ' 32 (20h) │  │
│         mov     al,00h                │[6]                 'W' 87 (57h) │  │
│         int     21h                   │[7]                 'o' 111 (6Fh)│  │
│ hello   endp                          │[8]                 'r' 114 (72h)│  │
│                                       │[9]                 'l' 108 (6Ch)│  │
│ cseg    ends                          │[10]                'd' 100 (64h)│  │
│                                       │[11]             '\x0D' 13 (0Dh) │  │
│                                       │[12]             '\x0A' 10 (0Ah) │  │
│ dseg    segment para 'DATA'           │─────────────────────────────────│  │
│                                       │byte ptr                         │  │
│ text    db      'Hello World',cr,lf   └─────────────────────────────────┘  │
└───────────────────────────────────────────────────────────────────────────┘
┌Watches────────────────────────────────────────────────────────────────────2┐
│                                                                            │
└───────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

配列

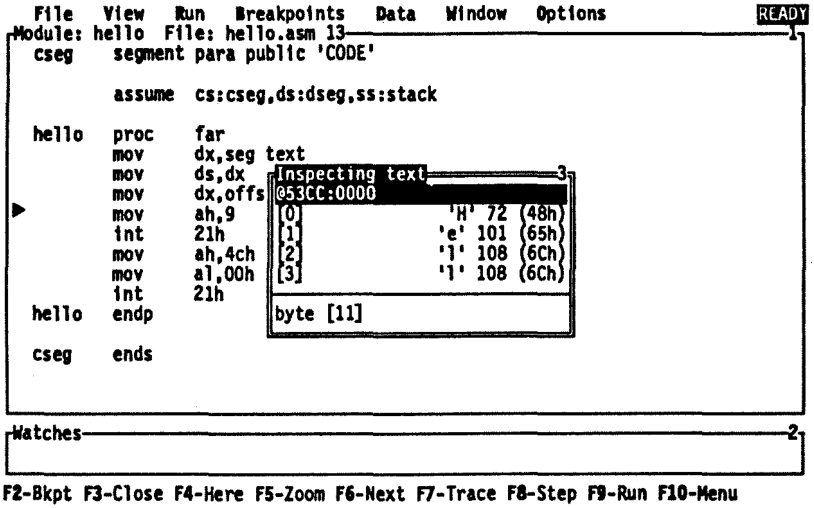
アセンブラの配列のInspectorウィンドウは，データの項目の配列の値を表示します。

```
WARRAY  DW 10 DUP (0)
MSG     DB "Greetings",0
```

配列の要素が各行に1つずつ表示されます。各行の左側には配列のインデックスが表示され，右側にはそのインデックスで指定される要素の現在の値が表示されます。その値がまた構造体のような複雑なデータ型のときは，できるだけ多くの要素の値が，丸カッコ( )で囲んで表示されます。

Rangeローカルコマンドを使うと，配列の一部分を調べることができます。多数の要素からなる配列の，中央部の要素を見たいときに便利です。

図6.16　アセンブラの配列のInspectorウィンドウ



構造体と共用体

アセンブラの構造体と共用体のInspectorウィンドウは，構造体と共用体のフィールドの

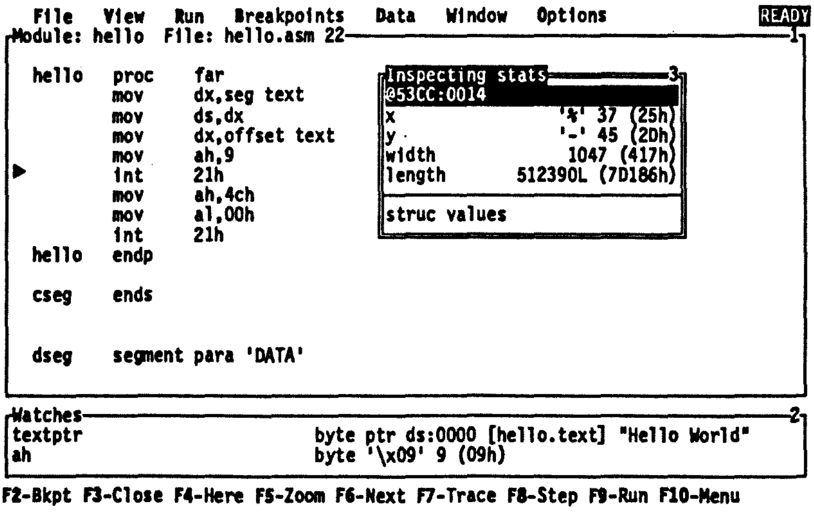
値を表示します。

```
X          STRUC
MEM1                DB        ?
MEM2                DD        ?
X          ENDS
ANX        X                  <1,ANX>

Y          UNION
ASBYTES             DB        10 DUP (?)
ASFLT               DT        ?
Y          ENDS
AY         Y                  <?,1,0>
```

このInspectorウィンドウには，フィールドの値を示すペインのに下に，もうひとつのペインがあります。このペインには，上のペインの中でハイライト表示されているフィールドのデータ型が表示されます。

図6.17　アセンブラの構造体のInspectorウィンドウ

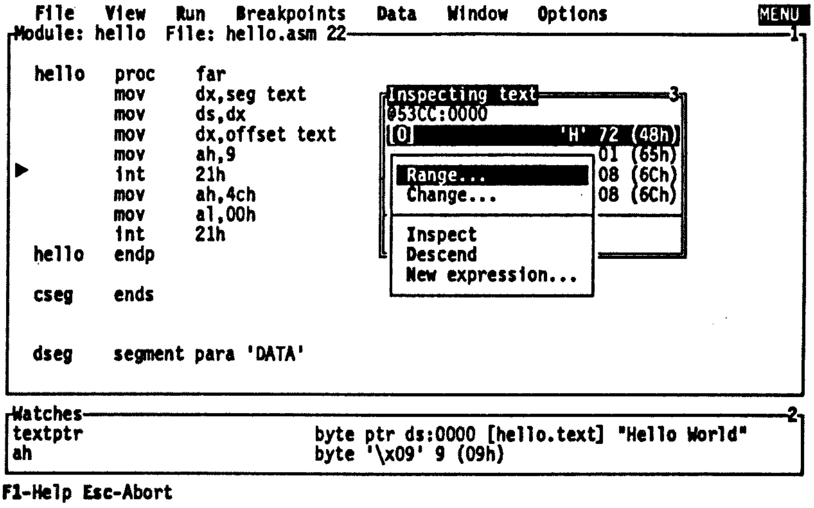


Inspectorウィンドウのローカルメニュー

このメニューのコマンドは，Inspectorウィンドウを非常に強力なものにしています。たとえばInspectローカルコマンドを選択することにより，もうひとつInspectorウィンドウを開いて，データ構造をたどっていくことができます。メニューの他のコマンドにより，ある範囲の値を調べたり，新しい変数を調べたりすることができます。

GRPH-F10を押すと，Inspectorウィンドウのローカルメニューが現われます。***CTRL***キーによるショートカットが有効なときは，***CTRL***キーとコマンドの先頭の文字を一緒に押すことにより，そのコマンドにアクセスすることができます。

図6.18　Inspectorウィンドウのローカルメニュー



Range

表示を開始する要素と，表示する要素の数を設定します。配列を調べていて，その配列の要素のうちの一部分だけを見たいときは，このコマンドを使ってください。

また長い配列の中央付近の少数の要素を見たいときは，このコマンドを使って，配列の

調べたい位置にInspectorウィンドウを開くことができます。

このコマンドは，Cのプログラムをデバッグしているときに特に便利です。なぜならCでは，ポインタを，char *pのように，1つの文字を指し示すように宣言しても，実際には，pによって指し示されるのはただ1文字だけでなく，文字の配列であることが多いからです。

Change

現在ハイライト表示されている変数の値を，あなたがプロンプトにしたがって入力した値に変更します。Turbo Debuggerは，使用している言語の代入演算子が変数のデータ型を変換するのとまったく同じように，必要な型キャスト（型変換）を行ないます。代入演算子とキャストについての詳しい説明は第9章を参照してください。

Inspect

現在ハイライト表示されている項の内容を示す，新しいInspectorウィンドウを開きます。このコマンドは，Inspectorウィンドウの中の項目が，構造体や配列のようにさらに複数の項目を含み，それらの項目を見たいときに便利です。

現在のInspectorウィンドウで調べているのが関数のときは，このコマンドを選択すると，新しく開いたウィンドウに，その関数のソースコードが表示されます。

このコマンドは，調べたい項目をハイライト表示させたあと**リターン**を押して選択することもできます。

ESCを押すと，新しいInspectorウィンドウが閉じ，ひとつ前のInspectorウィンドウに戻ります。データ構造を調べ終わった後，すべてのInspectorウィンドウを閉じるには，Window/Closeコマンドを使うか，そのショートカットである***F3***を押してください。

Descend

ハイライト表示された項の内容を，現在のウィンドウに表示します。新しいInspectorウィンドウを開かないこと以外は，Inspectコマンドと同じです。またNew expressionコマンドとInspectコマンドの組み合わせにも似ています。

注意：このコマンドでデータ構造をたどったときは，展開する前のレベルに後戻りすることはできません。複雑なデータ構造や，連結された長いリストをたどりたいとき，そしてデータ構造の前のレベルに後戻りしなくてもよいときは，このコマンドを使ってください。

このコマンドを使うと，画面のInspectorウィンドウの数を減らすことができます。

New expression

調べたい変数または式を入力するように求めてきます。新しいInspectorウィンドウは開きません。このコマンドにより，Inspectorウィンドウをそれ以上開かずに，別のデータを調べることができます。現在のInspectorウィンドウの中のデータを，そのまま表示しておく必要がないときは，このコマンドを使ってください。

第7章

ブレークポイント

Turbo Debuggerは，通常，ブレークポイント，ウォッチポイント，トレースポイントと区別して呼ばれているデバッガの機能をまとめて〝ブレークポイント〟と呼んでいます。

ブレークポイント，ウォッチポイント，トレースポイントは，習慣的に次のように定義されています。

ブレークポイントは，プログラム内の設定された場所でプログラムの実行を停止させます。

ウォッチポイントは，1命令ずつまたは1ソース行実行しながら，式の値がいつ真になるか監視します。

トレースポイントは，1命令ずつまたは1ソース行ずつ実行しながら，プログラム変数やメモリを参照する式の値がいつ変化するかウォッチします。

Turbo Debuggerは，ブレークポイントを設定するときに次の3つの要件を組み合わせて設定できるようにすることにより，上の3種類の機能を〝ブレークポイント〟というひとつの概念にまとめています。

■ブレークポイントが発生するプログラムの中の**ロケーション**

■ブレークポイントがトリガされる**条件**

■ブレークポイントがトリガされたときの**アクション**

〝ロケーション〟には，プログラムの単一の場所またはプログラム全体(グローバル)のどちらでも設定できます。グローバルに設定したときは，プログラムのすべてのソース行または命令において，ブレークポイント動作が発生する可能性があります。

〝条件〟には，次の3種類の条件を設定できます。

■必らず（無条件）

■式が真のとき

■特定のデータの値が変化したとき

また〝パスカウント〟を設定することもできます。〝パスカウント〟は，そのブレークポイントがトリガされ得るようになる前に〝条件〟が真にならなければならない回数です。

アクションには，次の3つのアクションを指定できます。

■プログラムの実行を停止する（ブレークポイント）
■式の値をログに書き込む
■式の実行（コードの接合）

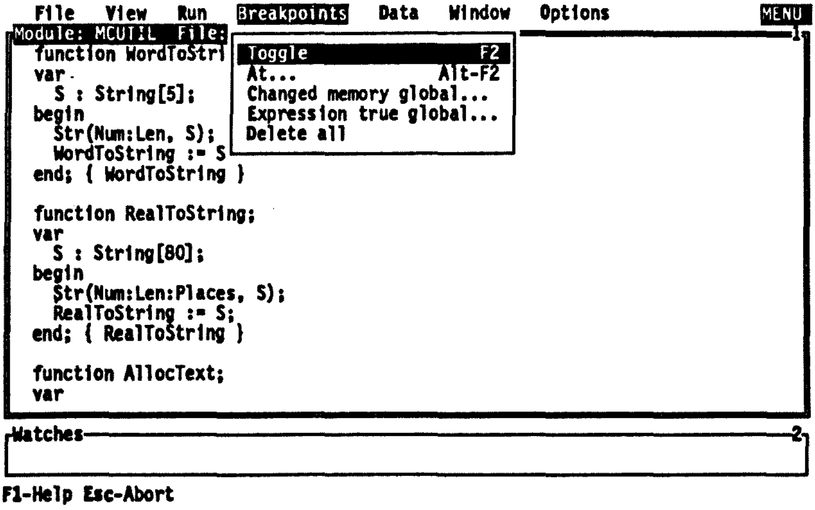
この章では，まずBreakpointsウィンドウとLogウィンドウについて説明します。次に単純ブレークポイント，条件ブレークポイント，プログラムの変数の値をログに記録するブレークポイントの設定方法について説明します。最後にプログラムの変数，式，または対象データの値がいつ変化するかを監視するブレークポイントの設定方法について説明します。

たいていの場合には，少数の単純ブレークポイントを設定して，プログラムの実行がそれらのロケーションのどれかに到達したら実行を停止させたいだけでしょう。このような場合には，ソースコードの行の上にカーソルを置いて**F2**キーを押すだけで，ソースコードのどの行でもブレークポイントを設定したり，解除したりすることができます。また，CPUウィンドウのCodeペインの中で命令を指定して**F2**キーを押すことにより，どの機械語コード行でもブレークポイントを設定できます。設定できるブレークポイントの数に制限はありません。

Breakpointsメニュー

ホットキー***GRPH-B***を押すことにより，Breakpointsメニューにアクセスすることができます。

図7.1　Breakpointsメニュー



Toggle

CPUウィンドウのCodeペインまたはModuleウィンドウの中の現在ハイライト表示されているアドレスに，ブレークポイントを設定します。そのアドレスにすでにブレークポイントが設定されているときは，そのブレークポイントを解除します。ホットキーは***F2***です。

At...

プログラム中の指定されたロケーションにブレークポイントを設定します。ブレークポ

イントを設定するアドレスの入力を求めます。ショートカットは***GRPH-F2***です。

Changed memory global...

メモリの指定された領域の値が変化したときにトリガされるブレークポイントを設定します。ウォッチするメモリの領域を入力するように求めます。詳しい説明は，この章の「**Breakpointsウィンドウのローカルメニュー**」の節のChanged memoryコマンドの説明を参照してください。

Expression true global...

指定された式の値が真になったときにトリガされるブレークポイントを設定します。式の入力を求めます。詳しい説明は，この章の「**Breakpointsウィンドウのローカルメニュー**」の中のCondition expression trueコマンドの説明を参照してください。

Delete all

設定されているすべてのブレークポイントを解除します。

ブレークポイント式に適用されるスコープ(適用範囲)

ブレークポイントのトリガ条件やアクションは，式によって制御することができます。この式は，プログラムが停止したロケーション（カレントロケーション）に適用されるスコープではなく，ブレークポイントが設定されているアドレスに適用されるスコープを使って評価されます。したがってブレークポイント式には，スコープオーバーライド機能によりスコープを変更しないかぎり，ブレークポイントを設定するアドレスにおいて有効な変数名しか使用できません。スコープについての詳しい説明は，第9章を参照してください。

ブレークポイント式中にルーチンのローカルな変数を使用すると，グローバル変数やモジュールローカルな変数を使用する場合に比べて，プログラムの実行速度が大幅に低下します。

Breakpointsウィンドウ

View/Breakpointsメインメニューコマンドを選択すると，Breakpointsウィンドウが開きます。このウィンドウにより，ブレークポイントをトリガする条件を調べたり，変更したりすることができます。またこのウィンドウを使って，新しいブレークポイントを追加したり，ブレークポイントを解除したり，設定済みのブレークポイントを変更することができます。

図7.2　Breakpointsウィンドウ

```
  File  View  Run  Breakpoints  Data  Window  Options                    READY
┌Module: MCALC  File: MCALC.PAS 122──────────────────────────────────────1┐
│ end; { Run }                                                           │
│                                                                        │
►begin                                                                   │
│   CheckBreak := False;                                                 │
│   SetColor(TXTCOLOR);                                                  │
│   ClrScr;          ┌Breakpoints───────────────────────────────────────3┐│
│   SetColor(MSGHEA ║Global 1          │ Breakpoint                      ║│
│   WriteXY(MSGHEAD ║MCALC.128         │ Data changed "CheckBreak" @5c95:c740║
│   SetColor(PROMPT ║MCOMMAND.136      │ Enabled                         ║│
│   WriteXY(MSGKEYP ║MCPARSER.330      │                                 ║│
│   GotoXY(80, 25); ║MCDISPLY.115      │                                 ║│
│   Ch := GetKey;   ║                  │                                 ║│
│   ClrScr;         └───────────────────────────────────────────────────┘│
│   InitVars;                                                            │
│   Changed := False;                                                    │
│   RedrawScreen;                                                        │
│   if (ParamCount > 0) then                                             │
│     LoadSheet(ParamStr(1));                                            │
└────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────2┐
│                                                                        │
└────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

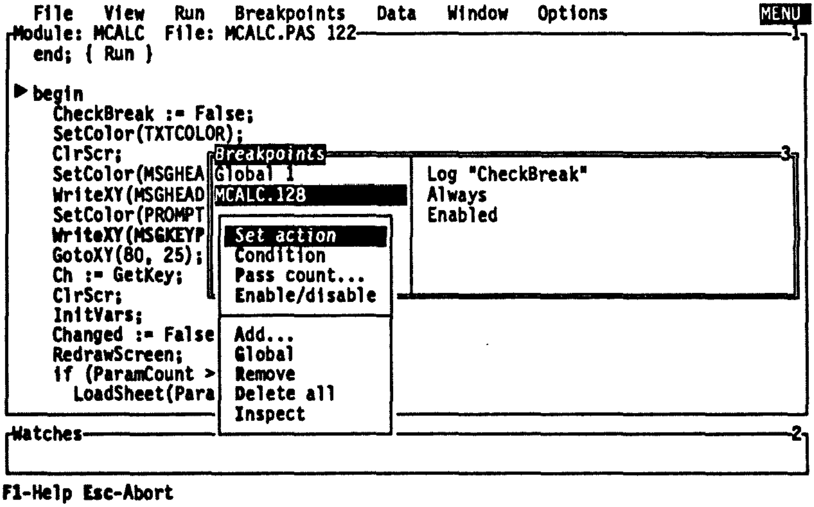
Breakpointsウィンドウには，2つのペインがあります。左側のペイン（ブレークポイントリスト）には，ブレークポイントが設定されているすべてのアドレスが表示されます。右側のペイン（ブレークポイントペイン）には，左側のペインのハイライト表示されているブレークポイントの詳しい情報が表示されます。ローカルメニューがあるのはブレークポイントリストペインだけです。***GRPH-F10***を押すとローカルメニューが表示されます。

Breakpointsウィンドウのローカルメニュー

このメニューの中のコマンドにより，新しいブレークポイントを追加設定したり，設定済みのブレークポイントを解除したり，ブレークポイントのアクションを変更したりすることができます。

GRPH-F10を押すと，Breakpointsウィンドウのローカルメニューが現われます。***CTRL***キーによるショートカットが使用可能に設定されているときは，***CTRL***キーとコマンドの先頭の文字を一緒に押すことにより，そのコマンドに直接アクセスすることができます。

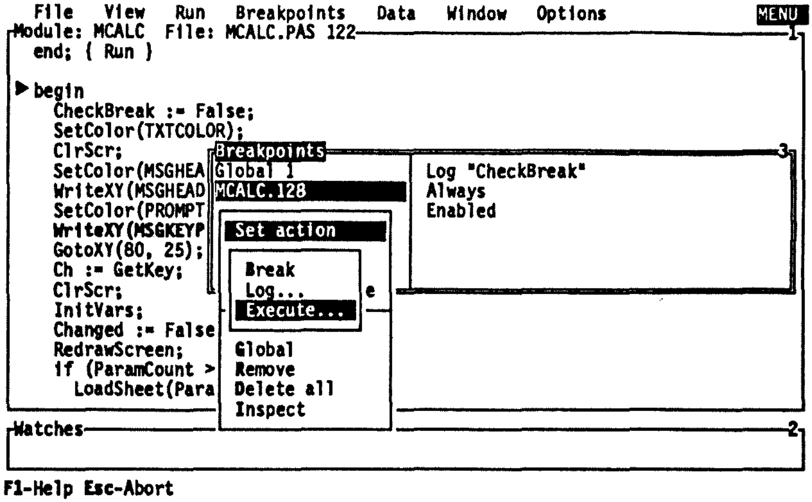
図7.3　Breakpointsウィンドウのローカルメニュー



Set action

ブレークポイントがトリガされたときのアクションを設定します。このコマンドは，次の図に示すメニューをポップアップします。

図7.4　Set actionメニュー



Break

ブレークポイントがトリガされたときにプログラムを停止させます。プログラムが停止すると，Turbo Debuggerの画面に戻ります。

Log

ブレークポイントがトリガされたとき，式の値をLogウィンドウに記録します。値をログに記録する式の入力を求めます。式が副作用を引き起こさないように注意してください。式とその副作用については第9章を参照してください。

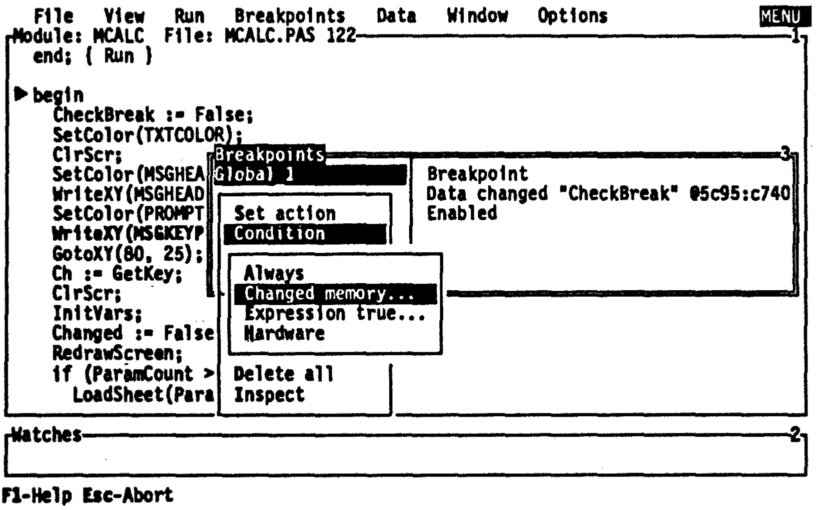
Execute

ブレークポイントがトリガされたときに式を実行します。実行する式の入力を求めます。式は，変数をある値に設定するなど，副作用を引き起こすものでなければなりません。このオプションを使うと，プログラムの希望するソース行または命令の前に式をひとつ挿入して実行する，コードの接合を行なうができます。

Condition

ブレークポイントがトリガされる条件を設定します。このコマンドは，図7.5に示すメニューをポップアップします。

図7.5　Conditionメニュー



Always

ブレークポイントが無条件でトリガされます。

Changed memory...

メモリ変数やその他のオブジェクトを監視（ウォッチ）し，オブジェクトの値が変化したときにブレークポイントをトリガ可能にします。監視したいオブジェクトを指し示す式と，監視したいオブジェクトの数を入力するように求めます。監視するメモリ領域の全バイト数は，式が指し示すオブジェクトのサイズ（バイト数）とオブジェクトの数の積になります。たとえばCのプログラムをデバッグしているときは，次のように入力します。

```
(long) a;4
```

この例ではウォッチする領域は16バイトです。なぜなら**long**型は**4**バイトで，それを4つ指定したからです。

グローバルブレークポイントにこの条件を付けると，プログラムの実行速度が大幅に遅くなります。なぜならソースコードが1行実行されるごとに，指定されたメモリ領域の内容が変化していないかチェックしなければならないからです。ハードウェアデバッガを使用すると，プログラムの実行速度がずっと速くなります。Changed memory型のブレークポイントがハードウェアにより支援されているときは，左側のペインの中のブレークポイント名の後に，アスタリスク*が表示されます。

単一アドレスのブレークポイントにこの条件を設定したときは，その行が実行されるたびに変数の値の変化がチェックされます。この場合には，グローバルブレークポイントを設定したときのような実行速度の低下はありません

Expression true

式が真（ゼロ以外の値）になったときに，ブレークポイントをトリガ可能にします。評価する式の入力を求めます。この式は，プログラムの実行がブレークポイントに到達するたびに評価されます。

Hardware

ハードウェアデバッガにより支援されたデバイスドライバにより，ブレークポイントがトリガされるようにします。386システムを持っていて，TDH386.SYSデバイスドライバを使っているとき，またはシステムにハードウェアデバッガボードを装備していて，ボードのメーカーからTurbo Debugger用のデバイスドライバが提供されたときは，このオプションを選択してください。

ハードウェアデバッガインターフェースと，このメニューから選択できるオプションの詳しい説明は，付録Fを参照してください。

Pass count...

ブレークポイントがトリガ可能になる前に，設定された条件が満足されなければならない回数（パスカウント）を設定します。パスカウントは，ブレークポイントに付けた条件が真のときにだけカウントダウンされます。したがってブレークポイントに条件とパスカ

ウントの両方を設定した場合には，設定した条件が設定した回数だけ真になったときにトリガ可能になります。

Enable/disable

現在ハイライト表示されているブレークポイントを，有効(Enable)または無効(Disable)にします。このコマンドはトグルになっていて，選択するたびにEnableとDisableの間で交互に切り替わります。無効にされたブレークポイントは，再び有効にするまであたかも削除されたかのように見えます。

このコマンドは，複雑なブレークポイントを設定して，今は使わないが後で再び使いたいときに便利です。ブレークポイントを削除したり，同じ条件，同じアクションで再設定しなくても済みます。

Add...

ブレークポイントのリストに，新しいブレークポイントを追加します。ブレークポイントを設定したいプログラム内のアドレスの入力を求めます。

グローバルブレークポイントを設定したいときは，このメニューのGlobalコマンドを使ってください。

このコマンドを選択しないで直接アドレスを入力することにより，Addコマンドを選択したときと同じプロンプトボックスが現われます。

ブレークポイントを追加したら，他のローカルメニューコマンドを使って，そのアクションを変更することができます。ブレークポイントは，追加したときは，パスカウントは1に，条件はalwaysに，アクションはbreak（停止）に設定されます。

Global

ブレークポイントのリストに，グローバルブレークポイントを追加します。1ソース行または1命令実行されるたびに，条件がチェックされます。ある変数の値がいつ変化するか知りたいときやある条件がいつ真になるかを知りたいときに，このコマンドを使います。

グローバルブレークポイントは，プログラムの実行速度を大きく低下させます。しかし，あるデータがプログラムの実行のどの時点でおかしくなるのかを調べるときなどには非常に便利です。

グローバルブレークポイントを追加したら，それをトリガする条件を設定しなければなりません。

Remove

現在ハイライト表示されているブレークポイントを解除します。

Delete all

グローバルブレークポイントも，個々のアドレスに設定されたブレークポイントも，すべてのブレークポイントを解除します。ブレークポイントでプログラムを停止させたいときは，あらためて設定しなければなりません。

Inspect

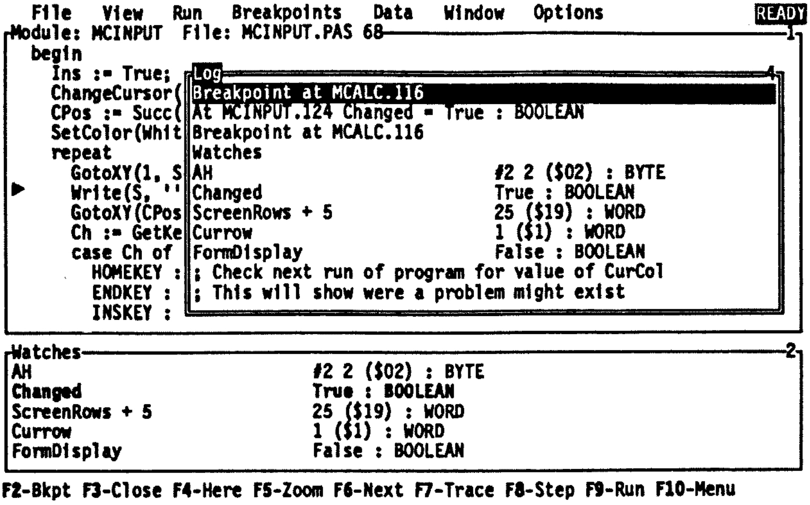
現在ハイライト表示されているブレークポイントに対応するソースコード行またはアセンブラ命令を表示します。そのブレークポイントが，プログラムのソース行に対応するアドレスに設定されているときは，Moduleウィンドウを開いてその行を表示します。そうでなければCPUウィンドウを開いて，Codeペインにそのブレークポイントが設定されている命令を表示します。

また希望するブレークポイントにハイライトバーを置いて**リターン**を押すことにより，このコマンドを選択することもできます。

Logウィンドウ

View/Logコマンドを選択すると，Logウィンドウが開きます。このウィンドウにより，デバッギングセッションで発生した重要な事象を，過去にさかのぼって調べることができます。

図7.6　Logウィンドウ



Logウィンドウは，このウィンドウに出力される事象のリストを表示します。表示されたリストは，新しい行が書き込まれるにつれてスクロールします。ログに50行以上の事象が書き込まれると，古い行がリストの先頭から失われます。ログの行数は，起動時にコマンドラインオプションで指定するか，インストールプログラム(TDINST)で設定することができます。

Open log fileローカルメニューコマンドを使うと，連続的にディスクファイルに書き込むことにより，ログをすべて保存することができます。

ログへの書き込みを引き起こす事象は，次のとおりです。

■指定したロケーションでプログラムが停止したとき。プログラムが停止したロケーションが，ログに記録されます。

■Addローカルメニューコマンドを選択し，ログに書き込むコメントを入力したとき入力したコメントが書き込まれます。

■式の値をログに記録するブレークポイントがトリガされたとき。式の値がログに書き込まれます。

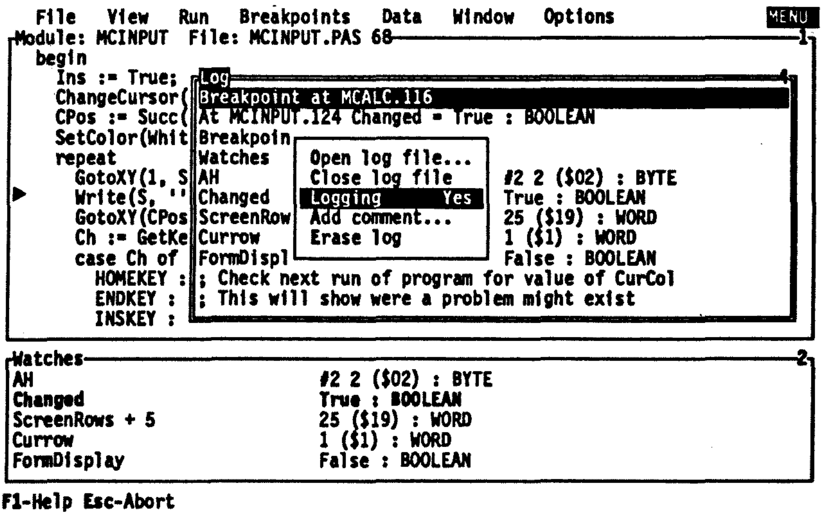
■Window/Dump pane to logコマンド(メインメニューバーから)を使って，ウィンドウのペインの現在の内容を記録したとき。

Logウィンドウのローカルメニュー

このメニューのコマンドは，ログのディスクファイルへの書き込み，ログへの記録の開始および停止，ログへのコメントの追加，ログの消去を行なうためのものです。

GRPH-F10を押すと，Logウィンドウのローカルメニューがポップアップします。***CTRL***キーによるショートカットが有効なときは，***CTRL***キーとコマンドの先頭の文字を一緒に押すことにより，そのコマンドに直接アクセスすることができます。

図7.7　Logウィンドウのローカルメニュー



Open log file

ログに書き込まれるすべての行をディスクファイルに書き込みます。ログを書き込むファイルの名前の入力を求めてきます。

ログファイルをオープンすると，Logウィンドウのリストに現在表示されている行は，すべてそのディスクファイルに書き込まれます。したがってLogウィンドウでディスクファイルに書き込みたいものが見つかったとき，ディスクログファイルをオープンして書き込むことができます。

ディスクログを開始したいが，現在Logウィンドウに表示されている情報は記録したくないときは，Open log fileコマンドを選択する前にErase log fileコマンドを選択して消去してください。

Close log file

Open log fileローカルメニューコマンドで指定されたファイルへのログ行の書き込みを停止し，ファイルをクローズします。

Logging

Logウィンドウへの書き込みの可/不可を切り替えます。

Add comment

ログへのコメントの挿入を可能にします。1行分のテキストの入力を求めます。

Erase log

ログリストを消去します。このコマンドを選択するとLogウィンドウの中の表示が消えます。ログのディスクファイルへの書き込みには影響を及ぼしません。

単純ブレークポイント

プログラムをデバッグしているときに普通最も行ないたいことは，特定のコードが実行される寸前に，プログラムの実行を停止させることです。

ブレークポイントをセットする方法はいろいろあります。各方法は，それぞれ異なる状況において便利です。

■Moduleウィンドウの中で希望するソース行に移動し，Breakpoints/Toggleコマンドを選択します(または***F2***を押します)。ブレークポイントがすでに設定されている行でこのコマンドを選択すると，そのブレークポイントが解除されます。

■CPUウィンドウのCodeペインの中で希望する命令に移動し，Breakpoints/Toggleコマンドを選択します(または***F2***を押します)。ブレークポイントがすでに設定されている行でこのコマンドを選択すると，そのブレークポイントが解除されます。

■Moduleウィンドウの中で希望するソース行に移動し，Breakpoints/Toggleコマンドを選択します(または***F2***を押します)。ブレークポイントがすでに設定されている行でこのコマンドを選択すると，そのブレークポイントが解除されます。

■Breakpoints/Atコマンドを選択し，ブレークポイントを設定したいコードアドレスを入力します。

■BreakpointsウィンドウのブレークポイントリストペインからAddローカルメニューコマンドを選択し，ブレークポイントを設定したいコードアドレスを入力します。

条件ブレークポイントとパスカウント

あるソース文が実行されるたびにブレークポイントがトリガされるのを望まない場合がしばしばあります。関心を持つ状況が発生する前に，その行のコードが何回も実行される場合には特にそうです。そこでTurbo Debuggerには，ブレークポイントが実際にいつトリガされるかを設定するための2つの手段（**パスカウント**と**条件**）が用意されています。

ある関数の10回目の呼び出しでプログラムを停止させたい場合，その関数の先頭にブレークポイントを設定し，さらにBreakpointsウィンドウのPass countローカルメニューコマンドを使って，そのブレークポイントが実際にトリガされる前にスキップさせたい回数を設定することができます。

あるロケーションで，ある条件が真のときに限りプログラムを停止させたい場合は，Condition/Expression trueローカルメニューコマンドを使って，式を設定することができます。この式は，プログラムの実行がブレークポイントに到達するたびに評価されます。式が真のとき（ゼロでないとき）は，ブレークポイントがトリガされます。

パスカウントと条件を組み合わせると，指定した条件式が指定した回数だけ真になった後で，はじめてブレークポイントがトリガ可能になります。

Condition/Changed memoryローカルメニューコマンドを使うと，指定したデータの値が変化したときにだけトリガされるブレークポイントを設定することができます。この方法は，何か注目しているものが正確にいつ変化するかを監視するグローバルブレークポイントよりもずっと効率的です。なぜならあるブレークポイントのところでだけウォッチすることで，必要な処理の量が大幅に減少するからです。

グローバルブレークポイント

どのソース行またはどの命令でもブレークポイントが発生し得るようにしたいときは，グローバルブレークポイントを使ってください。グローバルブレークポイントを設定する方法はいろいろあります。状況に応じて最も便利な方法を使ってください。

- ■Breakpointsウィンドウのaction listペインから，Globalローカルメニューコマンドを選択します。トリガ条件やパスカウントを設定したい場合，またはブレークポイントがトリガされたとき，実行を停止させるだけでなく何か他のことをさせたい場合に，この方法を使ってください。
- ■メインメニューバーから**Break points/Changed memory global**コマンドを選択します。メモリのある領域の内容が変化したとき実行を停止させたい場合に，この方法を使ってください。
- ■メインメニューバーから**Breakpoints/Expression true global**コマンドを選択します。式が真になったときに実行を停止させたい場合に，この方法を使ってください。

グローバルブレークポイントを設定したときは，Breakpointsウィンドウのローカルメニューコマンドを使って，条件やアクションを変更してください。そうしないと**Run/Trace into**メインメニューコマンドを使っているときと同じように，すべてのソース行でブレークポイント動作が起こることもあります。

1ソース行実行されるごとにグローバルブレークポイントを調べるには，メインメニューバーから**Run**コマンドの1つを選択して（または対応するファンクションキーにより）実行を再開するときに，カレントウィンドウがCPUウィンドウでないことを確認してください。

1命令実行されるごとにグローバルアクションを調べるには，プログラムを再実行するときに，カレントウィンドウがCPUウィンドウであることを確認してください。

データが変化したときにブレークする

あるデータがプログラムのどこで変えられるか見つけ出したいときは，まず前節で概説したテクニックのどれかを使ってグローバルブレークポイントを設定してください。次にActionウィンドウのdetailペインの中で，Condition/Changed memoryローカルメニューコマンドを選択することができます。そしてウォッチしたいメモリ領域を指定する式と，ウォッチする対象の数(必要であれば)を入力してください。

このコマンドを使ったときは，プログラムの実行速度が遅くなります。そのため，このテクニックを使ってデータが変化する正確なロケーションを見つけ出す前に，問題のありかを，ある程度絞り込んでおいてほうがいい場合もあります。

ハードウェアデバイスドライバがインストールされているときは，Turbo Debuggerは，データ領域の変化を監視するために，ハードウェアブレークポイントを設定しようと試みます。ハードウェアデバッガにより，サポートするハードウェアブレークポイントの数とタイプが異なります。***F10***/View/Breakpointsと選択してBreakpointsウィンドウを開くと，ブレークポイントがハードウェアを利用しているかどうかわかります。ハードウェアに支援されているブレークポイントには，傍らにアスタリスク(*)が表示されます。ハードウェアにより支援されたグローバルブレークポイントは，支援されないグローバルブレークポイントに比べてずっと高速です。

変数の値をログに書き込む

プログラムのある場所に到達するたびに，変数の値をログに記録しておくと便利な場合があります。どのような式の値でもログに記録することができます。たとえば，ある関数の呼び出しパラメータの値を記録すると，ログを調べることによって，その関数のどの呼び出しのときにパラメータに誤りがあったかを突き止めることができます。

変数の値をログに書き込むには，Breakpointsウィンドウのローカルメニューから**Set action/Log**コマンドを選択してください。そしてプロンプトにしたがって式を入力してください。複数の変数の値をログに記録したい場合は，複数のブレークポイントを設定しなければなりません。

式を評価する

ブレークポイントがトリガされるたびに副作用のある式を評価することにより，任意のソース行の前に，簡単に新しいコードを挿入することができます。この方法は，バグの修正をテストするときなどのように，ルーチンの動作を変えてみたいときに便利です。この方法を使うと，ルーチンに加えた僅かな変更をテストするために，コンパイル－リンクのサイクルをやり直さなくても済みます。

もちろんこの方法では，既存のコード行が実行される前にひとつの式を実行することしかできません。またこの方法は，既存のコード行を直接変更することもできません。

第8章

ファイルを調べる，変更する

Turbo Debuggerは，ディスクファイルを，デバッグしているプログラム上で自然な延長として扱えます。ディスクのどのファイルでも，ASCIIテキストとして，または16進数データとして表示させながら，調べたり変更したりすることができます。またTurbo Debuggerの内部から好みのワードプロセッサやテキストエディタを起動して，テキストファイルを編集することもできます。

この章は，2種類のディスクファイル(プログラムのソースコードを含むファイルとそれ以外のファイル）を調べたり，変更したりする方法について解説します。まずプログラムのソースファイルを調べたり変更したりする方法を説明します。次にその他のファイルを調べたり変更したりする方法を説明します。

プログラムのソースファイルを調べる

プログラムソースファイルとは，コンパイルされ，オブジェクトモジュール(.EXEファイル）が生成されるもとのファイルのことです。ソースファイルは，普通，プログラムのある部分の動作や構造を見たいときに調べます。またデバッグ中には，しばしば関数の引数が有効であることを確認したり，関数が正しい値を返しているかどうか調べるために，関数のソースコードを見ることが必要になります。

プログラムの部分的な実行を繰り返しているときでも，Turbo Debuggerは自動的にプログラムのカレントロケーション（実行中の番地）に対応するソースコードを表示します。

マクロ記号(#)で始まるコンパイラ指令(たとえばCの#include)やアセンブラの**INCLUDE**により，ソースファイルに取り込まれるファイルも，プログラムソースファイルと見なされます。プログラムソースファイルは，いつでも，Moduleウィンドウで見てください。なぜならModuleウィンドウで見ると，そのファイルがプログラムソースファイルであることをTurbo Debuggerに知らせることになるからです。Moduleウィンドウに表示させると，ファイルの中をあちこち動き回り，ブレークポイントを設定したり，プログラム変数の値を調べたり，いろいろなことができます。以下の節では，プログラムソースファイルを調べたり，変更したりする方法について説明します。

Moduleウィンドウ

メインメニューからView/Moduleコマンドを選択すると(またはショートカット***GRPH-F3***を押すと）Moduleウィンドウが開きます。

図8.1　Moduleウィンドウ

```
 File  View  Run  Breakpoints  Data  Window  Options                    PROMPT
┌Module: MCPARSER  File: MCPARSER.C 290═══════════════════════════════════════1┐
│ void reduce(int reduction)                                                   │
│ /* Completes a reduction */                                                  │
│ {                                                                            │
│  struct TOKENREC token1, token2;                                             │
│  int counter;                                          ┌Pick a module┐       │
│                                                        │MCALC        │       │
│  switch (reduction)                                    │MCDISPLY     │       │
│  {                                                     │MCINPUT      │       │
│   case 1 :                                             │MCOMMAND     │       │
│    token1 = pop();                                     │MCPARSER     │       │
│    pop();                                              │MCUTIL       │       │
│    token2 = pop();                                     └─────────────┘       │
│    curtoken.x.value = token1.x.value + token2.x.value;                       │
│    break;                                                                    │
│   case 2 :                                                                   │
│    token1 = pop();                                                           │
│    pop();                                                                    │
│    token2 = pop();                                                           │
└──────────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────────2┐
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
F1-Help ↑↓-Move ←┘-Select Letters-Match Esc-Abort
```

Moduleウィンドウが開くと，モジュールのリストが表示されます。このリストから，見たいモジュールを選ぶことができます。

Turbo Debuggerは，選択されたモジュールのソースファイルをロードします。ロードするソースファイルを見つけるために，Turbo Debuggerは，次のディレクトリを順番に探します。

1. コンパイラがそれらのソースファイルを見つけたディレクトリ
2. ***F10***/Options/Path for sourceコマンドまたは-**sd**コマンドラインオプションにより指定されたディレクトリ
3. カレントディレクトリ
4. デバッグしているプログラム（.EXEファイル）が入っているディレクトリ

Moduleウィンドウには，選択したモジュールのソースファイルの内容が表示されます。Moduleウィンドウのタイトルは，モジュールの名前とソースファイルの名前，それにカーソルが置かれている行の番号を示します。ウィンドウの1番左の桁の矢印→（IBM PCでは右向きの三角）は，カレントプログラムロケーションを示します。

Turbo Debuggerでデバッグするときは，.EXEファイルと元のソースファイルの両方が必要です。Turbo Debuggerは，ソースファイルを見つけるために，上記の1～4のディレクトリを順番に探します。

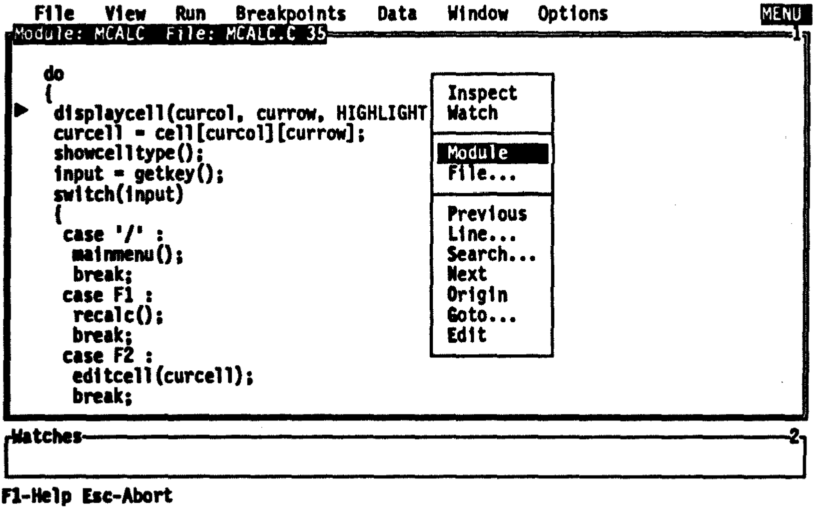
タイトルのファイル名の後にmodifiedと表示されたときは，そのファイルが，最後にコンパイルまたはリンクされて今デバッグしているプログラムが作成された後で，変更されたことを示します。この場合には，表示されているソースファイルは，今デバッグしているプログラムを作成するために使われたバージョンと，行番号が変わっている可能性があります。行番号が異なるときは，カレントプログラムロケーションを示す矢印が，間違った行に表示される可能性があります。

Moduleウィンドウのローカルメニュー

Moduleウィンドウのローカルメニューには，表示されているモジュールの中を動き回ったり，データを指定したり，指定したデータを調べたり，そのModuleウィンドウに新しいファイルやモジュールを表示させたりするための多数のコマンドがあります。

GRPH-F10を押すと，Moduleウィンドウのローカルメニューがポップアップします。また***CTRL***キーによるショートカットが有効なときは，***CTRL***キーとローカルメニューコマンドの先頭文字を一緒に押すことにより，そのコマンドに直接アクセスすることができます。

図8.2　Moduleウィンドウのローカルメニュー



Inspect

Inspectorウィンドウを開いて，現在カーソルが置かれているプログラム変数の値を示します。このコマンドは，呼び出す前に，ソースファイルの中の調べたいプログラム変数にカーソルを置いておかなければなりません。

また，Inspectコマンドを呼び出す前に，***INS***キーを使って調べたい式を指定することも

できます。この指定方法をとれば，ソースモジュールの中に見えている式をわざわざ入力しなくても済みます。

このように，Inspectコマンドを使うと変数名や式を入力しなくて済むので，プログラム変数の内容を調べるときには，このコマンドを多用することになるでしょう。

Watch

現在カーソルが置かれている変数を，Watchesウィンドウに付け加えます。このコマンドは，プログラムの実行中に変数の値を連続的に監視したい場合に便利です。このコマンドを行使する前に，ソースファイルの中の監視したいプログラム変数にカーソルを置いておかなければなりません。

また***INS***キーにより，監視したい式を指定することもできます。この指定方法によれば，ソースモジュールの中に見えている式をわざわざ入力しなくても済みます。

Module

表示されるモジュールのリストから希望するモジュールを選択することにより，別のモジュールを見るこができます。このコマンドは，現在のモジュールとは別のモジュールを見たいが，画面にそれ以上Moduleウィンドウを開きたくない場合に便利です。

複数のモジュールを同時に見たいときは，メインメニューからView/Another/Moduleを選択して，別のModuleウィンドウを開いてください。

File

現在表示されているモジュールを構成する他のソースファイルを見ることができます。表示されるファイルのリストから見たいファイルを選択してください。ほとんどのモジュールには，コードを含むソースファイルは1つしかありません。モジュールに含まれるその他のファイルは，通常，定数や構造体を定義しているだけです。モジュールのソースコードが複数のファイルにわかれている場合には，このコマンドを使ってください。

複数のファイルを同時に見たいときは，メインメニューバーからView/Another/Moduleと選択して別のModuleウィンドウを開くか，View/Fileと選択してFileウィンドウを開いてください。

Previous

カーソルを直前のロケーションに戻します。現在のモジュールの中を移動するコマンドを行使した後で，このコマンドを使って元のロケーションに戻ることもできます。

Line

カーソルをファイルの指定された行に移動します。移動したい行の行番号を入力してください。ファイルの最後の行の番号よりも大きな行番号を入力した場合は，カーソルはファイルの最後の行に置かれます。

Search

現在のカーソルの位置から，文字列を検索します。検索する文字列の入力を求めます。カーソルが変数名らしきものに置かれている場合には，検索プロンプトが現われたときに，その中にその変数名が入っています。***INS***キーを使ってファイルのブロックを指定したときは，検索プロンプトにブロックが入っています。この機能により，検索したい文字列が画面に表示されている文字列と同じ文字列であるときは，その文字列を入力しなくても済みます。

簡単なワイルドカード'?'と'＊'を使うこともできます。'?'は，その位置の文字がどんな文字でもよいことを示します。'＊'は，その位置に文字がなくてもよいし，どんな文字列でもよいことを示します。この検索は，ファイルの最後から先頭にラップアラウンド（ファイルの最後に到達したら先頭から検索を再開する）を行ないません。したがってファイルの全体を検索するには，***CTRL-ROLLDOWN***（IBM PCでは***Ctrl-PgUp***）を押してファイルの1行目に移動してください。

Next

Searchコマンドで指定した文字列の検索を繰り返します。このコマンドは，Searchコマンドで検索を行なった後でなければ使えません。

Searchコマンドで検索するとき，見つけ出したい文字列の前にある予期しない文字列が検索されることがあります。このコマンドを使うと，検索したい文字列を再入力しなくても，検索を繰り返すことができます。

Origin

カーソルを，カレントプログラムロケーション（CS:IP）に戻します。現在表示されているモジュールが，カレントプログラムロケーションを含むモジュールでないときは，Moduleウィンドウが切り替えられて，そのモジュールが表示されます。

このコマンドは，コードのあちこちを調べた後，プログラムが現在停止している場所に戻りたいときに便利です。

Goto

プログラム内の任意のロケーションに移動することができます。移動したいアドレスを入力してください。行番号，関数名，または16進アドレスを入力することができます。入力方法の詳しい説明は，第9章を参照してください。

移動先を示すラベルを直接入力すると，このコマンドを選択したときと同じプロンプトボックスが現われます。この方法は，このコマンドの便利なショートカットです。

Edit

エディタを起動します。エディタが起動されると，モジュールのソースファイルに変更を加えることができます。インストールプログラムTDINSTにより，エディタを起動するコマンドを設定することができます。

他のディスクファイルを調べる

Fileウィンドウを使うと，どのファイルでも，調べたり変更したりすることができます。Asciiコマンドまたは Hexコマンドを使うと，ファイルをASCIIテキストとしてまたは16進数データバイトとして表示させることができます。Asciiコマンドおよび Hexコマンドについては，この章の後の節で説明します。

Fileウィンドウ

メインメニューバーからView/Fileを選択すると，Fileウィンドウが開きます。DOS形式のワイルドカードを使ってファイルのリストを表示させそのリストから選択するか，またはファイル名を直接入力することにより，ファイルをロードすることができます。

図8.3　Fileウィンドウ

```
  File   View   Run   Breakpoints   Data   Window   Options              READY
┌Module: MCALC  File: MCALC.C 161──────────────────────────────────────────1┐
│ void main(int argc, char *argv[])                                         │
│ {  ┌File c:\tc\dos.h 197═══════════════════════════════════════════4┐    │
│  win│  char      _Cdecl peekb   (unsigned segment, unsigned offset);  │    │
│  ini│  void      _Cdecl poke    (unsigned segment, unsigned offset, int v│  │
│  ini│  void      _Cdecl pokeb   (unsigned segment, unsigned offset, char │  │
│  set│  int       _Cdecl randbrd (struct fcb *fcb, int rcnt);          │    │
│  set│  int       _Cdecl randbwr (struct fcb *fcb, int rcnt);          │    │
│  clr│  void      _Cdecl segread (struct SREGS *segp);                 │    │
│  wri│  int       _Cdecl setblock(unsigned segx, unsigned newsize);    │ER) │
│     │  int       _Cdecl setcbrk (int cbrkvalue);                      │    │
│  wri│  void      _Cdecl setdate (struct date *datep);                 │    │
│     │  void      _Cdecl setswitchar      (char ch);                   │    │
│  got│  void      _Cdecl settime (struct time *timep);                 │    │
│  get│  void      _Cdecl setvect (int interruptno, void interrupt (*isr) (│  │
│  set│  void      _Cdecl setverify        (int value);                 │    │
│  clr│  void      _Cdecl sleep   (unsigned seconds);                   │    │
│  ini│  void      _Cdecl sound   (unsigned frequency);                 │    │
│  mem└══════════════════════════════════════════════════════════════════┘   │
└───────────────────────────────────────────────────────────────────────────┘
┌Watches──────────────────────────────────────────────────────────────────2┐
│                                                                          │
└──────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Fileウィンドウには，選択したファイルの内容が表示されます。ウィンドウのタイトルには，現在表示されているファイルの名前が表示されます。ファイルがASCIIテキストとして表示されているときは，カーソルが置かれている行の行番号も表示されます。

Fileウィンドウを開いたときは，ファイルの内容がASCIIテキストであるかバイナリデータであるかにより（Turbo Debuggerがどちらであると判断するかにより），ASCIIテキストまたは16進数バイトとして表示されます。Display asコマンドを使うと，ASCIIから16進数バイトにまたはその逆に，いつでも表示を切り替えることができます。Display Asコマンドについては後で説明します。

図8.4　Fileウィンドウの16進数表示

```
  File   View   Run   Breakpoints   Data   Window   Options                READY
┌Module: MCALC  File: MCALC.C 161──────────────────────────────────────────1┐
│ void main(int argc, char *argv[])                                         │
│ {    ┌File c:\tc\dos.h──────────────────────────────────────────────4┐    │
│  win │00000: 2f 2a 09 64 6f 73 2e 68  /* dos.h                        │    │
│  ini │00008: 0d 0a 0d 0a 09 44 65 66       Def                        │    │
│  ini │00010: 69 6e 65 73 20 73 74 72  ines str                        │    │
│  set │00018: 75 63 74 73 2c 20 75 6e  ucts, un                        │    │
│  set │00020: 69 6f 6e 73 2c 20 6d 61  ions, ma                        │    │
│  clr │00028: 63 72 6f 73 2c 20 61 6e  cros, an                        │    │
│  wri │00030: 64 20 66 75 6e 63 74 69  d functi                        │ER) │
│      │00038: 6f 6e 73 20 66 6f 72 20  ons for                         │    │
│  wri │00040: 64 65 61 6c 69 6e 67 0d  dealing                         │    │
│      │00048: 0a 09 77 69 74 68 20 4d    with M                        │    │
│  got │00050: 53 44 4f 53 20 61 6e 64  SDOS and                        │    │
│  get │00058: 20 74 68 65 20 49 6e 74   the Int                        │    │
│  set │00060: 65 6c 20 69 41 50 58 38  el iAPX8                        │    │
│  clr │00068: 36 20 6d 69 63 72 6f 70  6 microp                        │    │
│  ini │00070: 72 6f 63 65 73 73 6f 72  rocessor                        │    │
│  mem └───────────────────────────────────────────────────────────────┘    │
└───────────────────────────────────────────────────────────────────────────┘
┌Watches───────────────────────────────────────────────────────────────────2┐
│                                                                           │
└───────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Fileウィンドウのローカルメニュー

Fileウィンドウのローカルメニューには，ディスクファイルの中を動き回ったり，ファイルの内容の表示形式を切り替えたり，ファイルに変更を加えたりするための多数のコマンドがあります。

GRPH-F10を押すと，Fileウィンドウのローカルメニューがポップアップします。***CTRL***キーによるショートカットが有効なときは，***CTRL***キーと希望するローカルメニューコマンドの先頭文字を一緒に押すことにより，そのコマンドに直接アクセスすることができます。

図8.5　Fileウィンドウのローカルメニュー

```
  File   View   Run   Breakpoints   Data   Window   Options                  MENU
 Module: MCALC  File: MCALC.C 161─────────────────────────────────────────────1
  void main(int argc, char *argv[])
  {   File c:\tc\dos.h──────────────────────────────────────────────────4
   win 00000: 2f 2a 09 64 6f 73 2e 68  /* dos.h
   ini 00                  65 66       Def
   ini 00  Goto            74 72  ines str
   set 00  Search          75 6e  ucts, un
   set 00  Next            6d 61  ions, ma
   clr 00                  61 6e  cros, an
   wri 00  Display as  Hex 74 69  d functi                                    ER)
       00  File...         72 20  ons for
   wri 00  Edit            67 0d  dealing
       00                  20 4d    with M
   got 00050: 53 44 4f 53 20 61 6e 64  SDOS and
   get 00058: 20 74 68 65 20 49 6e 74   the Int
   set 00060: 65 6c 20 69 41 50 58 38  el iAPX8
   clr 00068: 36 20 6d 69 63 72 6f 70  6 microp
   ini 00070: 72 6f 63 65 73 73 6f 72  rocessor
   mem
 Watches──────────────────────────────────────────────────────────────────────2

F1-Help Esc-Abort
```

Goto

ファイルの新しい行番号,またはオフセットに移行することができます。ファイルをASCIIテキストとして表示させているときは，行番号を入力してください。ファイルを16進数バイトとして見ているときは，ファイルの先頭からのオフセットを入力してください。オフセットを入力するときは，式を入力することができます。ファイルの最後の行を越える行番号，またはファイルエンドを越えるオフセットを入力したときは，ファイルの最後に移動します。

Search

現在のカーソルの位置から，文字列を検索します。検索する文字列の入力を求めます。カーソルが変数名らしきものに置かれている場合には，検索プロンプトが現われたときに，その変数名が入っています。***INS***キーを使ってファイルのブロックを指定したときは，検索プロンプトにそのブロックが入っています。この機能により，検索したい文字列が画面に表示されている文字列と同じ文字列であるときは，その文字列を入力しなくてもすみます。検索文字列の書式は，ファイルがASCIIまたは16進数バイトのどちらで表示されているかに

より異なります。

ファイルがASCIIで表示されているときは，簡単なワイルドカード'?'と'＊'を使うことができます。'?'はすべての文字にマッチします。'＊'はすべての文字列（文字数ゼロの空文字列を含む）にマッチします。

ファイルが16進数バイトで表示されているときは，バイト値の列からなるバイトリストか，式評価言語の規則（現在使用している言語の式の文法）にしたがって引用符で囲んだ文字列を入力してください。バイトリストの詳しい説明は第9章を参照してください。

検索は，ファイルの最後から先頭にラップアラウンド（ファイルの最後に到達したら先頭から検索を再開する）を行ないません。ファイルの全体を検索するには，***CTRL-ROLLDOWN***（IBM PCでは***Ctrl-PgUp***）を押して，ファイルの1行目に移動してください。

このコマンドは，検索したい文字列を直接入力することによって呼び出すこともできます。文字列を入力すると，Searchコマンドを選択したときと同じプロンプトボックスが現われます。

Next

Searchコマンドで指定した文字列の検索を繰り返します。このコマンドは，Searchコマンドで検索を行なった後でなければ使えません。

Searchコマンドで検索するとき，見つけ出したい文字列の前にある予期しない文字列が検索されることがあります。このコマンドを使うと，検索したい文字列が見つかるまで検索を繰り返すことができます。

Display as

ファイルの表示をASCIIテキスト形式から16進数バイト形式へ，またはその逆に交互に切り替えます。ASCII表示を選択したときは，ファイルは，エディタやワードプロセッサの画面で見るのと同じように見えます。Hex表示を選択したときは，各行の先頭に，その行に表示されているバイトのオフセットが16進数で表示されます。1行に表示されるデータは，ペインの幅によって変わり，8バイトまたは16バイトです。バイトの16進数表示の右側には，各バイトに対応する表示文字が表示されます。32よりも小さいバイト値と127よりも大きいバイト値に対しても，対応する記号が表示されます。このようにして，すべてのバイト値に対して対応する文字や記号が表示されます。

File

指定されたファイルを表示します。DOS形式のワイルドカードを使ってファイルのリストを表示させ，そのリストから選択するか，またはファイル名を入力することにより，見たいファイルをロードすることができます。2つの異なるファイル，または1つのファイルの2つの異なる部分を同時に見たいときは，View/Another/Fileコマンドでfileウィンドウをもう1つ開いてください。

Edit

ファイルをASCIIテキストとして見ているときは，TDINSTインストールプログラムを使って指定したエディタを起動します。エディタが起動されると，表示されているファイルを変更することができます。

ファイルを16進数のデータバイトとして見ているときは，エディタは起動しません。そのかわりに，カーソルの置かれているバイトと入れ替えるバイトを入力するように求めます。検索するためのバイトリストを入力するのとまったく同じように，バイトリストを入力してください。バイトリストの詳しい説明は第9章を参照してください。

第9章

式

式は，プログラムの中で使われているシンボル(変数，ルーチンの名前など)，定数，それにTurbo Debuggerがサポートする言語のひとつで定義されている演算子の組み合わせです。

Turbo Debuggerの式評価機能を利用すると，式を入力してその値を知ることができます。また式を使ってメモリ内の値を知りたいデータを指し示すこともできます。さらに値やメモリアドレスの入力を求めるプロンプトに対して式を入力することもできます（言語により式の文法が異なることに注意してください)。

メインメニューから**Data/Evaluate/Modify**を選択すると，式を入力して値を知ることができます。このコマンドは，プログラム内のデータの値を調べたり，また簡易電卓としても使うことができます。

この章では，Turbo Debuggerが式を評価するときにどのようにして正しい言語を選択するか，またどうすればTurbo Debuggerにある言語を使わせることができるかを説明します。

次にどの言語にも共通の式の構成要素について説明します。またプロセッサのレジスタにアクセスする方法について説明します。さらに各言語の式の構成要素（定数，プログラム変数，文字列，演算子など)について説明します。そして言語ごとに，Turbo Debuggerがサポートする演算子と，式の文法を表にして掲載します。

C, Pascal, アセンブラの式の詳細な説明は，それぞれTurbo C, Turbo Pascal, Turbo Assemblerのユーザーズガイドとリファレンスガイドを参照してください。

式を評価するための言語の選択

Turbo Debuggerは，通常，カレントモジュールのソースファイルの名前の拡張子に基づいて，使用する式評価機能と言語を決定します。カレントモジュールとは，プログラムが停止したモジュールです。

Options/Languageコマンドを使うと，式の評価のために使用する言語を選択できます。**C**, **Pascal**, または**Assembler**を選択すると，いつでもその言語が使用されます。**Source**オプションを選択すると，カレントモジュールのソースファイルの言語に戻すことができます（Turbo Debuggerがソース言語を確定できないときは，Cの文法を適用します)。

通常はTurbo Debuggerに，言語の選択をまかせたいでしょう。しかし自分で設定した方が都合が良いこともあります。たとえば，CまたはPascalのプログラムから呼び出されるアセンブラのモジュールをデバッグするときなどがそうです。式評価に使用する言語を指定すると，カレントモジュールの言語がそれと異なる場合にも，式中でその指定した言語でデータを参照するときはデータにアクセスすることができます。

コードアドレス，データアドレス，行番号

プログラムの変数やルーチンにアクセスしたいときは，通常その名前を入力します。また評価の結果がメモリアドレスを指し示す式を入力することもできます。さらに#123のように行番号の前に記号（#）を付けることにより，ソース行の番号でコードアドレスを指定することもできます。次の節では，カレントスコープの外にあるシンボルにアクセスする方法について説明します。

カレントスコープ外のシンボルへのアクセス

デバッガがシンボルを探す場所は，そのシンボルの〝スコープ（適用範囲）〟と呼ばれる範囲です。カレントスコープ外にあるシンボルにアクセスすることは，Turbo Debuggerを使う上で，ほとんどの場合に理解する必要のない高度な概念です。

通常Turbo Debuggerは，コンパイラと同じ方法方で式中のシンボルを探します。

たとえばCのコンパイラは，まずカレント関数の中の静的（関数内でローカル）シンボルを探し，次にカレントモジュールの中の静的（モジュール内でローカル）シンボルを探し，それからグローバルシンボルを探します。

Pascalは，まず現在の手続きまたは関数の中を探し，次に〝外側の〟サブプログラムの中を探し（アクティブなスコープが別のスコープの中にネストされているとき），その次に現在のユニットの実現部の中を探し（カレントスコープがユニットの内部にあるとき），最後にグローバルシンボルを探します。

Turbo Debuggerも，これと同じ順序でシンボルを探します。それでも見つからないときは，そのシンボルと一致する静的シンボルを見つけ出すために，他のすべてのモジュールを探します。そのためどのモジュールを探すか指定しなくても，他のモジュールの識別子を参照することができます。

Turbo Debuggerに，あるシンボルを見つけるためにどこか別のところを探させたい場合には，モジュールやモジュール内のファイルなどを指定することにより，そのシンボル名を探す場所を完全に制御することができます。この方法を使うと，プログラムのアドレスを割り当てられたすべてのシンボルにアクセスすることができます。ある関数または手続きの内部でだけ有効な，他に同名のシンボルがあるシンボルにもアクセスできます。

スコープをオーバーライドするための書式（文法）

シンボル名のスコープをオーバーライドするための方法は，どの言語に対しても同じです。

通常は，シャープ記号（#）を使ってスコープの要素を区切ります。使用している言語でまぎらわしくなければ，シャープ記号のかわりにピリオド（.）を使うこともできます。ピリオドを使うときは，一番最初のピリオドは省略することができます。

スコープをオーバーライドするための一般的な書式は次のとおりです（大カッコ［］で囲んだ項は，指定してもしなくてもかまいません）。

```
[#module[#filename]]#linenumber[#variablename]
```

または,

```
[#module[#filename]][#functionname]#vaiablename
```

モジュール（#module）を指定しないときは，カレントモジュールを指定したものとみなされます。

次にスコープオーバーライドを指定した正しいシンボル式の例を示します。スコープをオーバーライドするために使用できる要素の適正なすべての組み合わせに対して，1例ずつ示してあります。

次の6つの例は，行番号を使ってスコープをオーバーライドする方法を示しています。

#123　**カレントモジュールの行123**

#123#myvar1　**カレントモジュールの行123からアクセス可能なシンボルmyvar1**

#mymodule#123　**モジュールmymoduleの行123**

#mymodule#123#myvar1
　モジュールmymoduleの行123からアクセス可能なシンボルmyvar1

#mymodule#file1#123
　モジュールmymoduleの1部であるソースファイルfile1の行123

#mymodule#file1#123#myvar1
　モジュールmymoduleの1部であるソースファイルfile1の行123からアクセス可能なシンボルmyvar1

次の6つの例は，モジュール，ファイル，または関数名を使って，変数のスコープをオーバーライドする方法を示します。

#myvar2	#をつけないmyvar2と同じ
#myfunc#myvar2	ルーチンmyfuncからアクセス可能な変数myvar2
#mymodule#myvar2	モジュールmymoduleからアクセス可能な変数myvar2
#mymodule#myfunc#myvar2	モジュールmymoduleのルーチンmyfuncからアクセス可能な変数myvar2
#mymodule#file2#myvar2	mymoduleに含まれるファイルfile2からアクセス可能な変数myvar2
#mymodule#file2#myfunc#myvar2	mymoduleに含まれるファイルfile2の中で定義されているルーチンmyfuncからアクセス可能な変数myvar2

Turbo Debuggerは，次の書式のPascalのユニットオーバーライドもサポートします。

```
unitname.symbolname
```

式の評価のために使用される暗黙のスコープ

Turbo Debuggerが式を評価するときはいつでも，スコープオーバーライドが指定されていないシンボル名のために使用されるカレントスコープが，プログラムのどこであるか決定しなければなりません。スコープの決定は，多くの言語において重要です。なぜなら，関数または手続きの内部では，グローバルシンボルと同じ名前のシンボルを使うことができるからです。同名のシンボルが使用されているときは，Turbo Debuggerは，式中のシンボルがどちらのシンボルを指しているのか知らなければなりません。

Turbo Debuggerは，通常，カーソルの現在位置に基づいてスコープを決定します。したがってModuleウィンドウの中でカーソルを移動することにより，式の評価に適用されるスコープを選定することができます。

このことは逆に，プログラムが停止したカレントラインからカーソルを動かすと，式の評価結果が期待しているものと異なる可能性があることを意味します。式が確実に，プログラムのカレントスコープで評価されるようにしたい場合は，ModuleウィンドウのOriginローカルコマンドを使って，ソースコードのカレントロケーションに戻ってください。またCPUウィンドウのCodeペインの中で動き回ったり，Stackウィンドウの中のルーチンをカーソルで選択したり，Variablesウィンドウの中のルーチン名をカーソルで選択することにより，スコープを選定することもできます。

バイトリスト

コマンドには，バイトリストを入力するように求めるものがあります。ファイルを16進数形式で表示しているときのFileウィンドウのSearchローカルコマンドや，CPUウィンドウのDataペインの中のSearchおよびChangeローカルコマンドなどです。

バイトリストは，Option/Languageコマンドにより指定された言語の規則に適合するスカラー数（非浮動小数点数）と文字列の任意の組み合わせです。文字列もスカラーも，式中で使用されるのと同じ規則にしたがいます。

スカラーは，対応するバイト列に変換されます。たとえばPascalのlongint型の値123456は，4バイトの16進数の列56 34 12 00になります。

言語	バイトリスト	16進数データ
C	"ab"0x04"c"	61 62 04 63
Pascal	'ab'#4'c'	61 62 04 63
アセンブラ	1234"AB"	34 12 41 42

Cの式

Turbo Debuggerは，Cの式の規則を完全にサポートします。式はシンボル，演算子，文字列，変数，および定数を組み合わせたものです。これらの構成要素について，以下の節で説明します。

Cのシンボル

シンボルは，プログラムの中のデータやルーチンの名前です。シンボル名は，英字または下線記号（）で始まらなければなりません。2番目以降には，英字と下線記号に加えて，0から9までの数字を含むこともできます。シンボル名を入力するときは，先頭の下線記号を省略することができます。下線記号を付けないでシンボル名を入力して，その名前が見つからない場合には，先頭に下線記号を付け加えてもう一度探されます。コンパイラは通常，シンボル名の先頭に下線記号を付け加えるので，自分でシンボル名を入力するときは，下線記号を入力しなければなりません。しかしこの機能により，下線記号に気をつけなくても済みます。

Cのレジスタ疑似変数

Turbo Debuggerでは，Turbo Cコンパイラと同じように，疑似変数を使ってプロセッサのレジスタにアクセスすることができます。疑似変数は，プロセッサのレジスタに対応する変数名です。

疑似変数	型	レジスタ
_AX	unsigned int	AX
_AL	unsigned char	AL
_AH	unsigned char	AH
_BX	unsigned int	BX
_BL	unsigned char	BL
_BH	unsigned char	BH
_CX	unsigned int	CX
_CL	unsigned char	CL
_CH	unsigned char	CH
_DX	unsigned int	DX
_DL	unsigned char	DL
_DH	unsigned char	DH
_CS	unsigned int	CS
_DS	unsigned char	DS
_SS	unsigned char	SS
_ES	unsigned char	ES
_SP	unsigned int	SP
_BP	unsigned char	BP
_DI	unsigned char	DI
_SI	unsigned char	SI
_IP	unsigned int	命令ポインタ

次の疑似変数を使うと，80386のレジスタにアクセスすることができます。

疑似変数	型	レジスタ
_EAX	unsigned long	EAX
_EBX	unsigned long	EBX
_ECX	unsigned long	ECX
_EDX	unsigned long	EDX
_ESP	unsigned long	ESP
_EBP	unsigned long	EBP
_EDI	unsigned long	EDI
_ESI	unsigned long	ESI
_FS	unsigned int	FS
_GS	unsigned int	GS

Cの定数と数の表記

定数は，浮動小数点数または整数の両方が使えます。

整数定数は，数表記の設定を変更しないかぎり10進数とみなされます。Cで使用される数表記と基数は次のとおりです。

表記	基数
数字	10進数
0数字	8進数
0X数字	16進数
0x数字	16進数

整数型定数は，通常**int**型（16ビット）です。long型の定数（32ビット）を指定したいときは，その定数の後にlまたはLを付けなければなりません。たとえば123456Lのようにします。

浮動小数点型定数は，小数点を含む10進数で，普通の10進数表記または指数表記（科学的記数法）のどちらでも使用できます。

例： `1.234` `4.5e+11`

Cの文字列とエスケープシーケンス

文字列は，二重引用符("")で囲まれた1連の文字です。エスケープ文字として円記号(¥)を用いるCの標準的なエスケープシーケンスも使用できます。

文字列	値	文字
¥¥	0X5C	円記号
¥a	0X07	ベル
¥b	0X08	バックスペース
¥f	0X0C	フォームフィード
¥n	0X0A	改行
¥r	0X0D	復帰
¥t	0X09	水平タブ
¥v	0X0B	垂直タブ
¥xnn	nn	16進バイト値
¥nnn	nnn	8進バイト値

円記号の後に上の表に掲載されている文字以外の文字を続けたときは，その文字がそのまま文字列に挿入されます。

Cの演算子と演算子の優先順位

Turbo Debuggerは，Cと同じ演算子を，同じ優先順位でサポートします。またTurbo Debuggerには，C言語の演算子のセットには含まれていない，新しい演算子がひとつあります。それはダブルコロン演算子（：：）です。この演算子は，Cのどの演算子よりも高い優先順位を持ち，この演算子の前にある式と後にある式からfarアドレスを生成するために使用されます。たとえば次のように使用します。

```
0X1234::0X1000     _ES::_BX
```

次に示す1次演算子は，最も高い優先順位を持ちます。

```
( )    [ ]    .    ->    sizeof
```

結合規則は左から右です。

次に示す単項演算子は，1次演算子よりも低く，二項演算子よりも高い優先順位を持ちます。結合規則は右から左です。

```
*  &  -  !  ~  ++  --
```

二項演算子は，相互に次に示すような優先順位を持ちます。同じ行の演算子は，同じ優先順位を持ちます。

```
高     *   /   %
       +   -
       >>  <<
       <   >   <=  >=
       ==  !=
       &
       ^
       |
       &&
低     ||
```

三項演算子（?:）の優先順位は，どの二項演算子よりも下位です。

代入演算子（単純代入演算子と複合代入演算子）の優先順位は，三項演算子よりも低く互いに同等です。結合規則は右から左です。

```
=  +=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

式の中でCの関数を呼び出す

ソースコードの中で呼び出すのとまったく同じようにして,Cの式から関数を呼び出すことができます。Turbo Debuggerは，指定された関数引数の値を使って，プログラムのコードを実行します。これは関数の動作を素早くテストすることができる，非常に便利な方法です。引数の値を変えて関数を繰り返し呼び出し，戻り値が正しいかどうか調べることができます。

たとえば，プログラムの中に，ある数を別の数で累乗する（x^y）次のような関数が含まれているとき，

```
long power(int x, int y) {
    long temp = 1;
    while (y--)
        temp *= x;
    return(temp);
}
```

引数の値を変えてこの関数を呼び出し，実行した結果を示します。

Cの式	結果
power(3,2) * 2	18
25 + power(5,8)	390650
power(2)	エラー(引数が足りない)

副作用のあるCの式

評価されている間にデータの値を変えるCの式を評価するときは，副作用が発生します。副作用を避けたいときと，プログラム変数の値を意識的に変更するために副作用を生じさせたいときがあるでしょう。したがって，どのような場合に副作用が発生する可能性があるかを理解することは，非常に大切です。

代入演算子（=, +=など）は，その演算子の左側に書かれた変数の値を変えます。インクリメント（++）演算子とデクリメント（--）演算子は，前置演算子として使用されてい

るか後置演算子として使用されているかにより，その演算子の後または前の変数の値を変えます。

プログラムの1部分である関数を実行した場合には，もっと微妙な副作用が発生する可能性があります。たとえば，次のようなCの式を評価させた場合には，

```
myfunc(1,2,3) + 7
```

myfuncがプログラムの他の変数の値を変化させていると，その後のプログラムの動作がおかしくなる可能性があります。

Cのキーワードとキャスト（強制型変換）

Turbo Debuggerでは，Cプログラムの中で行なうのとまったく同じように，ポインタをキャストすることができます。キャストは，Cのデータ型の宣言をカッコで囲んだものです。キャストは，評価された値がメモリへのポインタになる式の前に置かなければなりません。

キャストは，ダブルコロン演算子(::)を使って生成したfarアドレスで指し示されるメモリロケーションの内容を調べたいときに便利です。

```
(long far *) 0x3456::0

(char far *) _ES::_BX
```

キャストを使うと，データ型についての情報が作成されていないプログラム変数にアクセスすることができます。デバッグ情報を生成せずにモジュールをコンパイルしたときに便利な手法です。変数のデータ型がわかっているときは，再コンパイルして再リンクしなくても，データ型をカッコで囲んで変数の名前の前に置くだけで，データ型を指定することができます。

たとえば変数*iptr*が整数を指し示すポインタのとき，次のようなCの式を評価することより，*iptr*が指し示す整数を調べることができます。

```
* (int *) iptr
```

Turbo Debuggerのためにキャストするときは，次のようなCのキーワードを使うことができます。

char	float	far
int	double	huge
short	unsigned	struct
long	near	union
		enum

Pascalの式

Turbo Debuggerは，Pascalの式の規則をサポートします。ただし，文字列（連結）演算子と集合演算子はサポートしません。式は，演算子，文字列，変数，および定数を組み合わせたものです。これらの構成要素について，以下の節で説明します。

Pascalのシンボル

Pascalのシンボルは，プログラムの中のデータやルーチンに，ユーザがつけた名前です。Pascalのシンボル名は，英字（*a*～*z*，*A*～*Z*）および下線記号（_）で始まらなければなりません。2番目以降には，英字と下線記号に加えて0から9までの数字を含むこともできます。

通常シンボルは，Pascalのスコープ規則にしたがって評価されます。ネストされたローカルシンボルは，同じ名前の他のシンボルをオーバーライドします。カレントスコープ外のシンボルにアクセスしたいときは，このスコープの適用をオーバーライドすることができます。詳しい説明は，少し前の「**カレントスコープ外のシンボルへのアクセス**」の節を参照してください。

Pascalの定数と表記

定数は，浮動小数点数と整数のどちらでも使用できます。負の定数は，前にマイナス符号（-）をつけます。小数点またはe（その後の数が指数であることを示す）を含む数は浮動小数点数です。

例：　`123.4`　　`456e34`　　`123.45e-5`

整数型定数は通常10進数です。16進数であることを示すには，先頭にドル記号（$）をつけます。整数型定数は，10進数では-2,147,483,648から2,147,483,647の範囲，16進数では$00000000から$FFFFFFFFの範囲の数でなければなりません。

Pascalの文字列

文字列は，1連の文字を単引用符で囲んだものです。

```
'abc'
```

10進数で表わしたコントロールキャラクタ（制御文字）コードの前に記号（#）をつけることにより，コントロールキャラクタを文字列の中に埋め込むことができます。たとえば次のようにします。

```
'def#7xyz'
```

Pascalの演算子

Turbo Debuggerは，Pascalの式演算子をすべてサポートします。
単項演算子は，最も高い優先順位を持ち，互いに優先順位の高低はありません。

@	識別子のアドレスを取り出す
^	ポインタの指し示すメモリロケーションの内容を取り出す
not	各ビットの値を反転する
型名	型キャスト
＋	単項プラス，正
－	単項マイナス，負

二項演算子は，単項演算子よりも優先順位が低く，互いに優先順位の高低があります。優先順位の高いものから順番に示します。同じ行の演算子の順位は同等です。

```
*     /     div   mod   and   shl   shr
in    +     -     or    xor
<     <=    >     >=    =     <>
```

代入演算子：=の優先順位は最下位です。この演算子は，Cの代入演算子と同じように値を返します。

Pascalの関数と手続きの呼び出し

式の中でPascalの関数や手続きを参照することができます。

たとえば，プログラムの中で，次のような整数を2で割る関数*HalfFunc*を宣言しているものとします。

```
function HalfFunc(i:integer) : real;
```

その場合には，**Data/Evaluate/modify**コマンドを選択し，次のように入力して*HalfFunc*を呼び出すことができます。

```
HalfFunc(3)
HalfFunc(10)=HalfFunc(10 div 2)
```

また手続きを呼び出すこともできます。ただし手続きは値を返さないので，式の中で呼び出すことはできません。

手続きまたは関数の名前だけを入力すると，Turbo Debuggerは，そのアドレスと宣言を表示します。呼び出しパラメータを持たない関数や手続きを呼び出すときは，その関数またはシンボル名の後に1対のカッコを置きます。たとえば

```
MyProc()        MyProcを呼び出す
MyProc          MyProcのアドレス，その他を得る
MyFunc=5        MyFuncのアドレスを5と比較する
MyFunc()=5      MyFuncを呼び出し，戻り値を5と比較する
```

アセンブラの式

Turbo Debuggerは，アセンブラの式の規則をすべてサポートします。式は，演算子，文字列，変数，および定数を組み合わせたものです。これらの構成要素について，この節で説明します。

アセンブラのシンボル

シンボルは，プログラムの中のデータやルーチンにユーザがつけた名前です。アセンブラのシンボル名は，英字（*a*～*z*，*A*～*Z*）または4つの記号（@ ? _ $）のどれかで始まらなければなりません。2番目以降には，英字とこれらの記号に加えて，0から9までの数字を含むこともできます。またピリオド（.）も，シンボル名の先頭の文字として使用できます。ただし名前の中には使用できません。

記号$は，レジスタペアCS:IPに示されるカレントプログラムロケーションを表わす特別な記号です。

アセンブラの定数

数は，浮動小数点数または整数のどちらでも使用できます。浮動小数点定数は，小数点を含む10進数で，普通の10進数表記または指数表記（科学的記数法）のどちらでも使用できます。

例： `1.234`　　`4.5e+11`

整数型定数は，他の数表記に変更しないかぎり，16進数とみなされます。

表記	基数
数字	16進数
数字O	8進数
数字Q	8進数
数字D	10進数
数字B	2進数

数は0～9の数字で始まらなければなりません。英字*A*～*F*で始まる16進数を入力したいときは，その前にゼロ（0）をつけてください。

アセンブラの演算子

Turbo Debuggerは，アセンブラの演算子をほとんどサポートします。サポートする演算子を，優先順位の高い順に示します。

```
xxx PTR (BYTE PTR...)
.    (構造体の要素の参照)
:    (セグメントオーバーライド)
OR   XOR
AND
NOT
EQ   NE   LT   LE   GT   GE
+    -
*    /    MOD  SHR   SHL
単項+   単項-
OFFSET   SEG
( )   [ ]
```

変数の値は，代入演算子（=）を使って変えることができます。

```
a = [BYTE PTR DS:4]
```

書式制御

Turbo Debuggerは，式の値をそのデータ型に適した書式で表示します。デフォルトの表示書式を変更したいときは，式の後にカンマを置き，繰り返し回数（オプション）と書式文字（オプション）を指定してください。繰り返し回数は，ポインタと配列に対してだけ指定することができます。式のデータ型に対して不適切な書式を指定した場合には，指定は無視されます。次の表に書式文字を掲載します。

文字	書式
c	文字または文字列をそのまま表示します。表示できない文字値は，エスケープ形式または数表記で表示されます。
d	整数を10進数として表示します。
f [#]	指定した桁数の浮動小数点形式で表示します。桁数を指定しないときは，必要な桁数が使用されます。
m	メモリを参照する式の値を16進数バイトとして表示します。
md	メモリを参照する式の値を10進数バイトとして表示します。
p	ポインタの値をそのまま表示します。可能なときはセグメントをレジスタ名で示します。またポインタが指し示す対象も表示します。この書式は，書式制御が指定されないときのデフォルトです。
s	文字配列または文字配列へのポインタを，引用符で囲んだ文字 列として表示します。この文字列は，最後にヌル文字がつけられます。
x または **h**	整数を16進数として表示します。

第10章

アセンブラレベルのデバッグ

この章は,アセンブリ言語による80x86ファミリプロセッサのプログラミングに精通している人々のためのものです。ほとんどのプログラムのデバッグでは，この章に説明されている機能を使う必要はありません。しかしこの章に説明されている手法を使うと，より容易に見つけ出すことができる問題もあります。

まずソースデバッグでは十分でない場合を説明します。次に逆アセンブラとアセンブラが組み込まれたCPUウィンドウについて説明します。さらにデータをそのまま16進数バイトとして調べる方法や変更する方法，関数の呼び出しスタックを調べる方法，CPUレジスタを調べる方法と変更する方法，そして最後にCPUフラグを調べる方法とそれらを変更する方法を説明します。

ソースデバッグでは十分でないとき

プログラムをデバッグしている時間の大部分は，ソースレベルでプログラムのデータやコードを参照したり，ソースコードのあちらこちらを部分的に実行したりすることに費やされます。

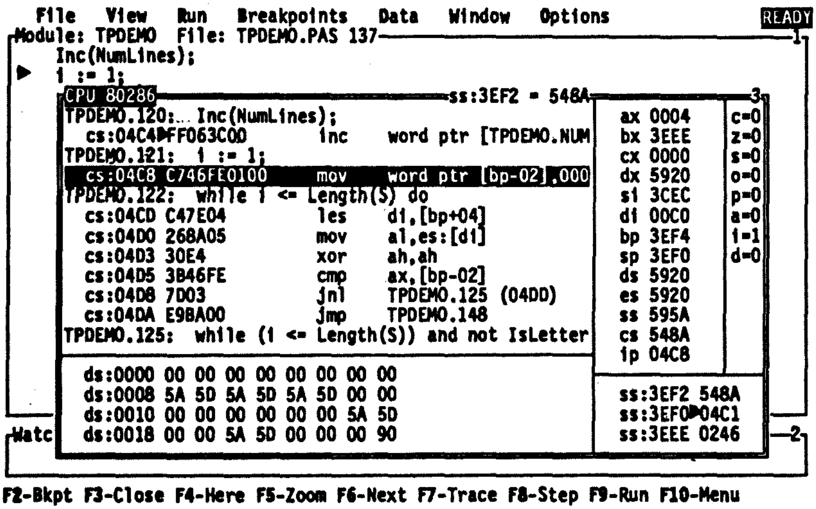
しかしコンパイラが生成した命令，CPUレジスタの内容，スタックの内容などを見ることにより，問題をつきとめることができる場合もあります。このような手法を使うには，80x86ファミリのプロセッサの動作と，コンパイラによるソースコードの機械語命令への翻訳の仕方を，よく知っていなければなりません。CPUの内部動作についてはすぐれた書籍が多数あるので，ここでは詳しく説明しません。またコンパイラの翻訳の仕方は，ソースコードの各行から生成された命令を見ることにより，すぐに理解できるようになるでしょう。

たとえばCでは，いろいろな動作をするソースコードを1行に書くことができます。ところがデバッガは，Cの式ではなく，ソース行ごとに実行を制御するので，ソース行のある1部分を実行した中間結果を知りたいことがあります。そういうときは，機械語命令を1命令ずつ実行することにより，その中間結果を知ることができます。ただしそれには，コンパイラがソース文をどのように機械語命令に翻訳したかを理解するための多少の努力が必要です。

CPUウィンドウ

CPUウィンドウは，CPUの状態をすべて表示します。またプログラムのコードやデータを構成するビットやバイトを調べたり，変更したりすることができます。Codeペインでは，組み込みのアセンブラを使って，アセンブラのソース命令を入力するのとまったく同じやり方で命令を入力することにより，プログラムを1時的にパッチすることができます。またどのようなデータ構造でも，それを構成するバイトにアクセスし，いろいろな形式で表示させたり，変更したりすることができます。

図10.1　CPUウィンドウ



メインメニューバーからView/CPUコマンドを選択すると，CPUウィンドウが開きます。新しいCPUウィンドウには，カレントウィンドウで見ているものに応じて，コード，データ，またはスタックの適切な部分が表示されます。これは，現在カーソルが置かれているコード，データ，スタックの状態を見るための便利な方法です。次の表は，CPUコマンドを選択したときにカーソルが置かれる位置を示します。

カレントウィンドウ	CPUウィンドウペイン	カーソルが置かれる位置
Stackウィンドウ	Stack	カレントSS:SP
Moduleウィンドウ	Code	カレントCS:IP
Actionウィンドウ	Code	アクションアドレス
Variableウィンドウ	Data *	その項のアドレス
Inspectorウィンドウ	Data	その項のアドレス
Breakpointsウィンドウ（グローバルでないとき）	Code	ブレークポイントのアドレス

* カレントウィンドウに表示されているのがルーチンのときはCodeペイン

CPUウィンドウペインの一番上の行には，プロセッサの種類(8086,80186,80286,80386)が表示されます。CPUウィンドウペインには5つのペインがあります。あるペインから別のペインに移るには，***TAB***または***SHIFT-TAB***を押してください。左上のペインは(Codeペイン）は，ソース行と逆アセンブルしたアセンブリ命令を，織り交ぜて表示します。その右隣のペイン(Regisiterペイン)は，CPUレジスタの内容を表示します。右上のペイン(Flagsペイン)は，8つのCPUフラグの状態を表示します。左下のペイン(Dataペイン)は，選択したメモリ領域の内容が16進数バイトで表示されます。右下のペイン（Stackペイン）は，スタックの内容が表示されます。

Codeペインの中では，矢印→が，カレントプログラムロケーション（*CS:IP*）を示します。Stackペインの中では，矢印→が現在のスタックポインタの値（*SS:SP*）を示します。またStackペインでは，表示されている値の上に，新しい値を直接入力することができます。

Codeペインの中でハイライト表示されている命令が，メモリのある場所を参照しているときは，そのメモリアドレスとそのアドレスの内容が，CPUウィンドウの最上行に表示されます。これにより，命令のオペランドが指し示すメモリの場所と，次に読み出される値または上書き変更される値を見ることができます。

Flagsペインは，各CPUフラグの値を表示します。次の表に，Flagsペインに表示されるフラグと，それらのフラグを表わす記号を示します。

ペイン内の文字	フラグ名
c	キャリー
z	ゼロ
s	符号
o	オーバーフロー
p	パリティ
a	補助キャリー
i	割り込み可
d	方向

GRPH-F10を押すと，Codeペインのローカルメニューがポップアップします。また***CTRL***キーによるショートカットが有効なときは，***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのローカルコマンドを使うことができます。

Codeペイン，Dataペイン，Stackペインの中では，***CTRL*-左矢印**および***CTRL*-右矢印**を押すと，そのペインの表示開始アドレスが，1バイトずつ上または下にシフトします。この方法は，表示を少しだけ調節したい場合には，Gotoコマンドを使うよりも簡単です。

Codeペイン

指定されたアドレスの機械語命令を逆アセンブルして表示します。Mixedローカルコマンドは，次の3つの表示モードを順番に切り替えます。

No　ソースコードは表示されません。逆アセンブルされた命令だけが表示されます。

Yes　ソースコード行が表示され，その下にそのソース行に対応する逆アセンブルされた命令が表示されます。カレントモジュールが高級言語のソースモジュールのときは，この表示モードに設定されます。

Both　逆アセンブルされた行を，対応するソースコード行で置き換えます。対応するソースコードがないときは，逆アセンブルされた命令が表示されます。アセンブラモジュールをデバッグしていて，逆アセンブルされた命令のかわりにもとのソースコード行を見たいときは，このモードを使ってください。カレントモジュールがアセンブラのソースモジュールのときは，この表示モードに設定されます。

逆アセンブルされた行の左側の部分には，その命令のアドレスが表示されます。アドレスは，セグメントとオフセットが16進数で表示されます。セグメントの値がCSレジスタの現在の値に等しいときは，セグメント値のかわりにレジスタ名CSが表示されます。CPUウィンドウの幅が十分に広いとき（ズームされているか，大きさが再設定されているとき）は，命令を構成するバイトの値が表示されます。逆アセンブルされた命令はその右側に表示されます。

逆アセンブラ

Codeペインは，デバッグしているプログラムを，自動的に逆アセンブルして表示します。オペランドのアドレスが，グローバルシンボル，静的シンボル，または行番号のどれかに対応するときは，Mixed表示モードがYesに設定されている場合には，そのアドレスのかわりに対応するシンボルまたは行番号が表示されます。またアドレスの内容が行番号のときは，その行番号の後に対応するソース行が表示されます。

グローバルシンボルは，シンボル名だけが表示されます。静的シンボルは，モジュール

名，その後に記号（#）または（.），その後にシンボル名という形式で表示されます。行番号は，モジュール名，その後に記号（#）または（.），その後に10進数で表記した行番号という形式で表示されます。

即値オペランドが表示されているときは，その桁数からバイト数を推測することができます。バイト即値は2桁，ワード即値は4桁です。

Turbo Debuggerは，8087/80287/80387数値演算コプロセッサの命令を検出し，それらの命令を逆アセンブルします。

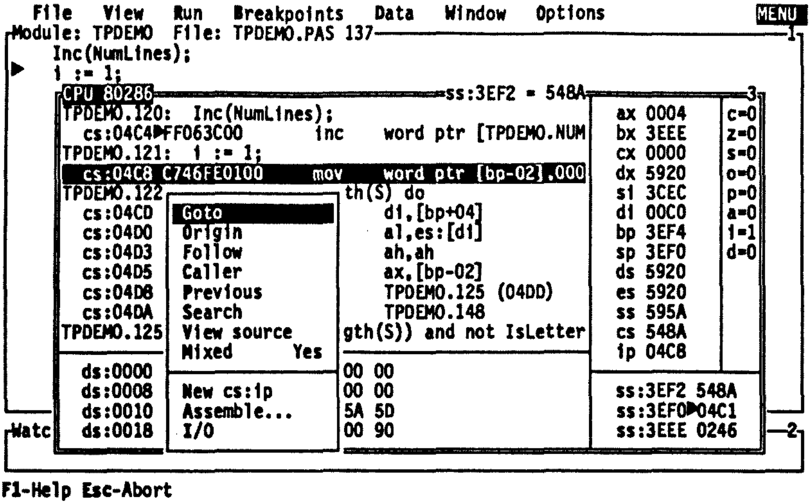
命令ニーモニック**RETF**は，この命令がfarリターン命令であること示します。通常の**RET**ニーモニックは，nearリターン命令を示します。

可能な場合には，**JMP**命令と**CALL**命令の目的アドレスが，行番号で表示されます。*CS:IP*の指し示す命令が**JMP**または条件ジャンプ命令のときは，その命令が実行されるとジャンプを引き起こす場合に限り（JMPはいつでも），ジャンプの方向を示す矢印（↑または↓）が表示されます。また**MOV**，**ADD**，その他の命令により使用されるメモリアドレスは，シンボリックアドレス（レジスタ名を使ってシンボル表示されたアドレス）で表示されます。

Codeペインのローカルメニュー

Codeペインにいないときは，***TAB***または***SHIFT-TAB***を押してCodeペインに移行してください。それから***GRPH-F10***を押して，ローカルメニューをポップアップさせてください。

図10.2　Codeペインのローカルメニュー



Goto

移動したい新しいアドレスを入力するように求めます。デバッグしているプログラムの外部のアドレスも，入力することができます。したがってBIOS ROM，DOS，常駐ユーティリティなどのコードを調べることができます。アドレスの入力方法についての詳しい説明は第9章を参照してください。

Previousコマンドは，Codeペインの表示を，Gotoコマンドを使用する前の場所に戻します。

Origin

Codeペインの表示を，CS:IPレジスタが示すカレントプログラムロケーションに戻します。このコマンドは，プログラムの実行を停止させていろいろな部分を調べた後，カレントプログラムロケーションに戻りたいときに便利です。

Previousコマンドは，Codeペインの表示を，Originコマンドを使用する前の場所に戻します。

Follow

Codeペインの表示を，現在ハイライト表示されている命令の目的アドレス（その命令が制御を移すアドレス）に移動します。条件ジャンプ命令に対しては，ジャンプが起きたかのように，ジャンプ先のアドレスを表示します。

このコマンドは，**CALL**，**JMP**，条件ジャンプ（**JZ**，**JNE**，**LOOP**，**JCXZ**，その他），およびINT命令に対して使用できます。

Previousコマンドは，Codeペインの表示を**Follow**コマンドを選択する前の場所に戻します。

Caller

Codeペインの表示を，現在の割り込みルーチンまたはサブルーチンを呼び出した命令に移動します。

このコマンドは，いつでも働くわけではありません。割り込みルーチンまたはサブルーチンがデータをスタックにプッシュしてしまったときは，Turbo Debuggerが，そのルーチンがどこから呼び出されたのかわからない場合があります。

Previousコマンドは，Codeペインの表示を，**Caller**コマンドを選択する前の場所に戻します。

Previous

Dataペインの表示を，表示アドレスを変えるコマンドにより変える前のアドレスに戻します。矢印キーや***ROLLUP***および***ROLLDOWN***キーを使って表示アドレスを変えたときは，変更前の表示位置は記憶されません。

Previousを選択したときは，Codeペインのその時点の表示位置が記憶されます。そのため**Previous**を選択すると，2つの表示アドレスに交互に切り替わります。

Search

検索したい命令またはバイトリストの入力を求めます。命令を入力するときは，Assembleコマンドを使って入力するときと同じように入力してください。

検索しようとする命令に注意してください。メモリのどこに向けてアセンブルされるかにより，アセンブルされるバイト値が変化しない命令だけを，検索してください。たとえ

ば，次の命令の検索は問題ありません。

```
PUSH    DX
POP     [DI+4]
ADD     AX,100
```

しかし次の命令を検索しようとすると，結果はどうなるかわかりません（下の例ではオペランドがロケーション（番地）になっている命令のとき）。

```
JE      123
CALL    MYFUNC
LOOP    $-10
```

命令のかわりにバイトリストを入力することもできます。バイトリストの入力方法は第9章を参照してください。

Mixed

逆アセンブルされた命令とソースコードを異なる組み合わせで表示する3つの表示モードを順番に切り替えます。

No ソースコードは表示されません。逆アセンブルされた命令だけが表示されます。

Yes ソースコード行が表示され，その下にそのソース行に対応する逆アセンブルされた命令が表示されます。カレントモジュールが高級言語のソースモジュールのときは，この表示モードに設定されます。

Both 逆アセンブルされた行を，対応するソースコード行で置き換えます。対応するソースコードがないときは，逆アセンブルされた命令が表示されます。アセンブラモジュールをデバッグしていて，逆アセンブルされた命令のかわりにもとのソースコード行を見たいときは，このモードを使ってください。カレントモジュールがアセンブラのソースモジュールのときは，この表示モードに設定されます。

New CS:IP

プログラムロケーションカウンタ（*CS:IP*）を，現在ハイライト表示されているアドレスに設定します。プログラムを再実行すると，このアドレスから実行が始まります。プログラムのある部分を，実行しないでスキップしたいときに便利です。

このコマンドは，細心の注意を払いながら使ってください。CS:IPをカレントロケーショ

ンとスタックの状態が異なる別のロケーションに設定すると，プログラムはほとんど確実に暴走します。ですからCS:IPは，カレントルーチン外のアドレスに設定しないでください。

Assemble

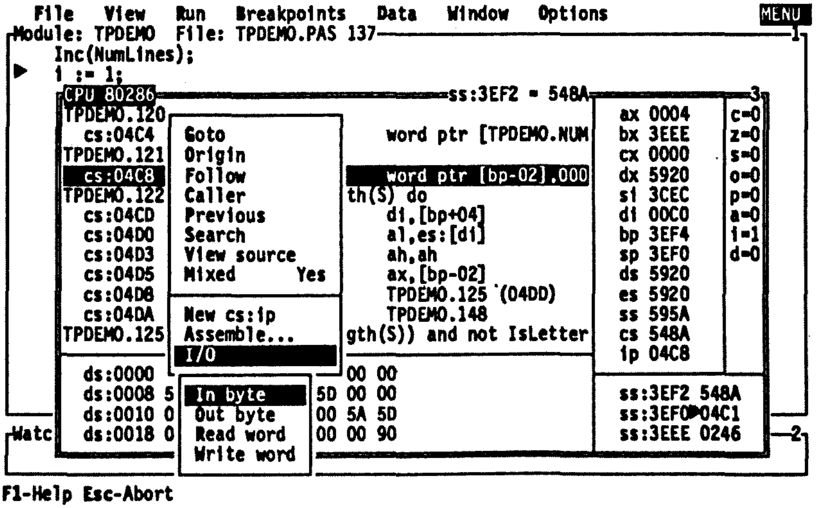
命令をアセンブルし，現在ハイライト表示されているロケーションの命令と置き換えます。アセンブルする命令の入力を求めます。詳しい説明は，この章の「**アセンブラ**」の節を参照してください。

アセンブルしたい命令を直接入力すると，Assembleコマンドを選択したときと同じプロンプトボックスが現われます。

I/O

CPUのI/O空間からの読み込み，同空間への書き込みを行ないます。カード上のI/Oレジスタを内容を調べたり，そこに値を書き込んだりすることができます。このコマンドを選択すると，図10.3に示すようなローカルメニューがポップアップします。

図10.3　I/Oメニュー



In byte

I/Oポートから1バイト読み込みます。値を調べたいI/Oポートを入力するように求めます。ワードサイズのI/Oポートから読み出すには，Read wordオプションを使用してください。

Out byte

I/Oポートに1バイト書き込みます。書き込みたいI/Oポートと，書き込みたい値を入力するように求めます。ワードサイズのI/Oポートに書き込むには，Write wordオプションを使用してしてください。

Read word

I/Oポートから1ワード読み込みます。値を調べたいI/Oポートを入力するように求めます。バイトサイズのI/Oポートから読み込むには，In byteオプションを使用してください。

Write word

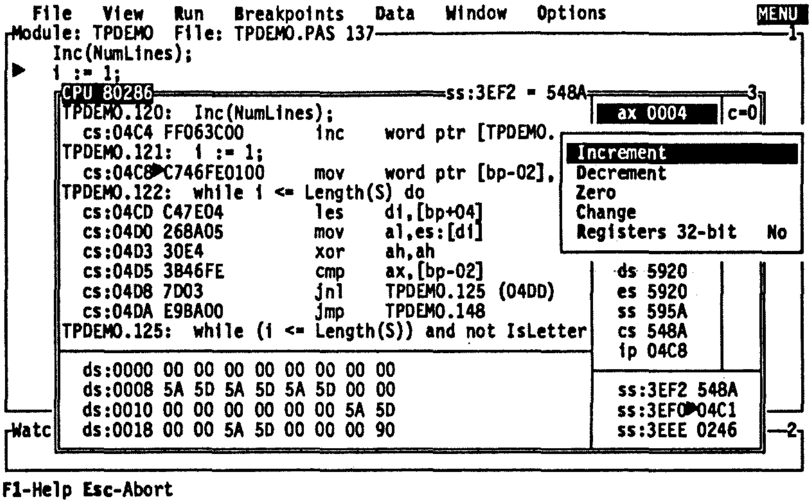
I/Oポートに1ワード書き込みます。書き込みたいI/Oポートと，書き込みたい値を入力するように求めます。バイトサイズのI/Oポートに書き込むには，Out byteオプションを使用してしてください。

これらのコマンドを使うときは，注意してください。I/Oデバイスの中には，そのポートが読み込まれると，ステータスビットをリセットしたり，ポートに新しいデータをロードしたりするものがあるからです。

Registerペインのローカルメニュー

GRPH-F10を押すと，Registerペインのローカルメニューがポップアップします。またCTRLキーによるショートカットが有効になっているときは，***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることもできます。

図10.4　Registerペインのローカルメニュー



Increment

現在ハイライト表示されているレジスタの値に1を加えます。このコマンドは，レジスタの値を僅かに変更するときに便利です。たとえば〝1だけずれている〞バグを修正する場合などです。

Decrement

現在ハイライト表示されているレジスタの値から1を引きます。

Zero

現在ハイライト表示されているレジスタの値をゼロにします。

Change

現在ハイライト表示されているレジスタの値を変更します。新しい値の入力を求めます。新しい値を入力するときは，式評価機能をフルに利用することができます。

このコマンドは，レジスタにセットしたい新しい値を直接入力することにより起動することもできます。入力を始めると，ローカルメニューでChangeコマンドを選択したときと同じプロンプトボックスが現われます。

Registers 32-bit

CPUレジスタの表示を，16ビット値表示と32ビット値表示に交互に切り替えます。80386プロセッサ上では，このコマンドを使って16ビットレジスタ表示にセットしないかぎり32ビットレジスタとして表示されます。386チップの32ビットアドレシング機能を使うプログラムをデバッグしている場合に限り，32ビットレジスタを見る必要があります。16ビットアドレシングだけを使う普通のプログラムをデバッグしている場合には，16ビットレジスタ表示を選択することができます。

Flagsペインのローカルメニュー

GRPH-F10を押すと，Flagsペインのローカルメニューがポップアップします。また，***CTRL***キーによるショートカットが有効になっているときは，***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることもできます。

図10.5　Flagsペインのローカルメニュー

```
  File  View  Run  Breakpoints  Data  Window  Options                MENU
 Module: TCDEMO  File: TCDEMO.C 41──────────────────────────────────────1
        while (readaline() != 0) {
 ►                wordcount = makeintowords(buffer);
  CPU 80286──────────────────────ss:FFC0 = 57B3──────────────────3
  TCDEMO#41:  nwords += wordcount;                 ax 0001   c=0
    cs:0227 037EFA          add    di,[bp-06]      bx 0A4B
  TCDEMO#42:  totalcharacters += analyzewords(buffer);  cx 0874   Toggle
    cs:022A B87408          mov    ax,0874         dx 0A24
    cs:022D 50              push   ax              si 0000   p=0
    cs:022E E87400          call   TCDEMO#analyzewords  di 0000   a=0
    cs:0231 59              pop    cx              bp FFC6   i=1
    cs:0232 0146FC          add    [bp-04],ax      sp FFBC   d=0
    cs:0235 1156FE          adc    [bp-02],dx      ds 5A51
  TCDEMO#43:  nlines++;                            es 5A51
    cs:0238 46              inc    si              ss 5A51
  TCDEMO#39:  while (readaline() != 0) {           cs 554A
                                                   ip 021C
    ds:0000 00 00 00 00 54 75 72 62     Turb       ss:FFBE 0051
    ds:0008 6F 2D 43 20 2D 20 43 6F o-C - Co       ss:FFBC►09B8
    ds:0010 70 79 72 69 67 68 74 20 pyright        ss:FFBA 0246
 W  ds:0018 28 63 29 20 31 39 38 37 (c) 1987                     ──2
 F1-Help Esc-Abort
```

Toggle

フラグの値を1のときは0に，0のときは1にセットします。値0は〝クリア〟を，1は〝セット〟を示します。また**リターン**を押すと，現在ハイライト表示されているフラグの値を反転することができます。

Dataペイン

このペインは，指定されたメモリの領域の内容をそのまま表示します。各行の最も左側の部分には，その行に表示されているデータのアドレスが表示されます。アドレスは，16進数表示のセグメントとオフセットとして表示されます。セグメント値が現在のDSレジスタの値に等しいときは，セグメント値がレジスタ名DSで置き換えられます。

その右の部分には1つまたは複数のデータが表示されます。この部分の表示形式は，Display asローカルメニューコマンドで選択されている表示モードにより異なります。浮動小数点表示形式（Comp, Float, Real, Double, Extended）のどれかを選択すると，浮動小数点数が各行に1つずつ表示されます。Byte形式では各行に8バイトずつ表示されます。Word形式では各行に4ワードずつ表示されます。Long形式では各行に2ロングワードずつ表示されます。

各行の最も右側の部分には，表示されているデータバイトに対応する表示文字が表示されます。Turbo Debuggerは，すべてのバイト値を，対応する表示文字で表示します。ですから16進数ダンプ領域の右側に変わった記号が表示されても驚かないでください。これらの記号は16進数バイト値に対応する表示文字です。

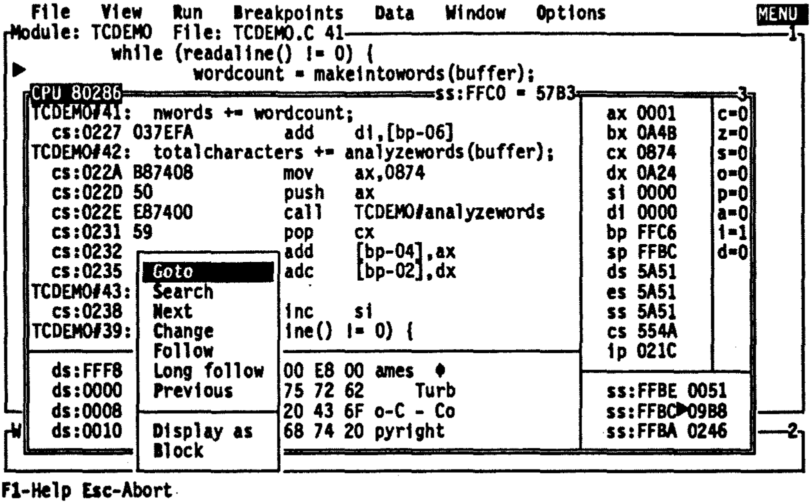
各行に表示されるバイトの数は，Display asコマンドによりセットされた表示形式により異なります。

注意：Dataペインを使って，表示メモリ，ROM BIOS用データ領域，または下位メモリのベクタなどの内容を調べる場合には，Turbo Debuggerが動作しているときにメモリにある実際の値ではなく，デバッグしているプログラムが動作するときにそこに存在する値が表示されます。ですから表示される値は，それを見ているときにそのメモリ領域にある値と同じではありません。Turbo Debuggerは，自分も使用しているメモリ領域にユーザがアクセスしようとしたときは，プログラムのその領域の値を退避している場所から正しいデータ値を読み出して表示します。

Dataペインのローカルメニュー

GRPH-F10を押すと，Dataペインのローカルメニューがポップアップします。また，***CTRL***キーによるショートカットが有効になっているときは，***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることもできます。

図10.6　Dataペインのローカルメニュー



Goto

指定されたメモリ領域のデータを表示します。表示したい新しいアドレスを入力してください。DOSの内部や，常駐ユーティリティの中のアドレスなど，プログラムの外部のアドレスも入力できます。したがってBIOSのデータ領域のデータを調べることもできます。アドレスの入力の仕方の詳しい説明は，第9章を参照してください。

Search

文字列を検索します。カーソルの位置により示されるカレントメモリアドレスから，検索を開始します。検索したいバイトリストを入力してください。検索は，セグメントの最後に達するとそこで停止します。

バイトリストの説明は，第9章を参照してください。

Next

Searchコマンドで指定されたバイトリストの，次の出現を検索します。

Change

現在のカーソル位置のバイトを変更することができます。表示がASCII形式またはバイト形式の場合には，バイトリストの入力を求めます。それ以外の場合には，現在の表示形式にしたがって入力するように求めます。バイトリストの説明は，第9章を参照してください。

新しい値を直接入力すると，ローカルメニューでChangeコマンドを選択したときと同じプロンプトボックスが現われます。

Follow

ワードポインタ（near－オフセットのみ）の指し示す先をたどることができます。Dataペインの表示を，現在カーソルが置かれているワードにより指定されるオフセットに移動します。

Long follow

ロングポインタ（far－セグメントおよびオフセット）の指し示す先をたどることができます。Dataペインの表示を，現在カーソルが置かれている2つのワードにより指定されるオフセットに移動します。

Previous

Dataペインの表示を，表示アドレスを変えるコマンドにより変更する前のアドレスに戻します。矢印キーや***ROLLUP***および***ROLLDOWN***キーを使って表示アドレスを変えたときは，変更前の表示位置は記憶されません。

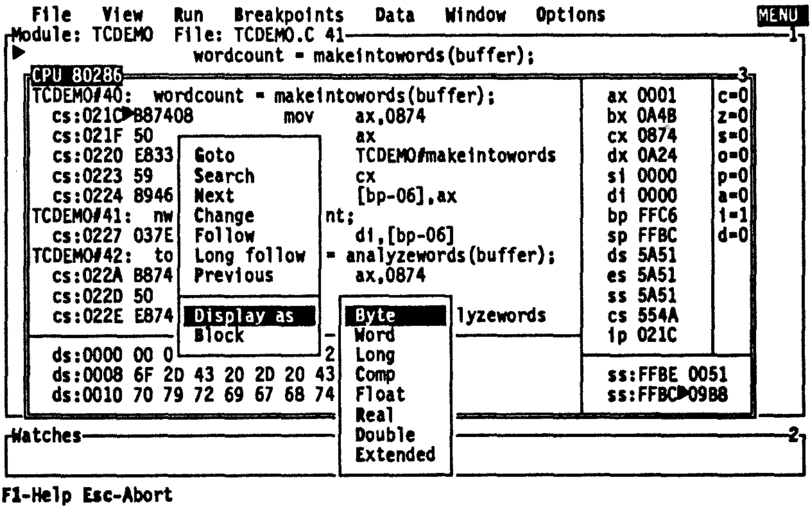
Turbo Debuggerは，最近の5つのアドレスをスタックに記憶します。したがってFollow，Long follow，Gotoコマンドを複数回使用したあとで，それを逆にたどることもできます。

Display as

Dataペインにデータを表示する形式を選択することができます。C，Pascal，アセンブラにより使用されるすべてのデータ形式を選択することができます。図10.7に示すメニュー

から選択してください。

図10.7　Display asメニュー



Byte

1バイト長の16進数として表示するようにセットします。

Cの**char**型，Pascalのbyte型に対応します。

Word

ワード長の16進数として表示するようにセットします。2バイトの16進数値が表示されます。

Cの**int**型，Pascalのword型に対応します。

Long

ロング長の16進数として表示するようにセットします。4バイトの16進数値が表示されます。

Cの**long**型，Pascalのlongint型に対応します。

Comp

8バイト長の整数を表示するようにセットします。整数の10進数値が表示されます。
Pascalのcomp（IEEE）型です。

Float

浮動小数点数として表示するようにセットします。浮動小数点数値が，科学的記数法で表示されます。
C の**float**型，Pascalのsingle型（IEEE）と同じです。

Real

Pascalの6バイト長の浮動小数点数を表示するようにセットします。浮動小数点数値が，科学的記数法で表示されます。
Pascalのreal型です。

Double

8バイト長の浮動小数点数を表示するようにセットします。浮動小数点数値が，科学的記数法で表示されます。
Cのdouble型，Pascalのdouble型（IEEE），アセンブラのTBYTE型と同じです。

Extended

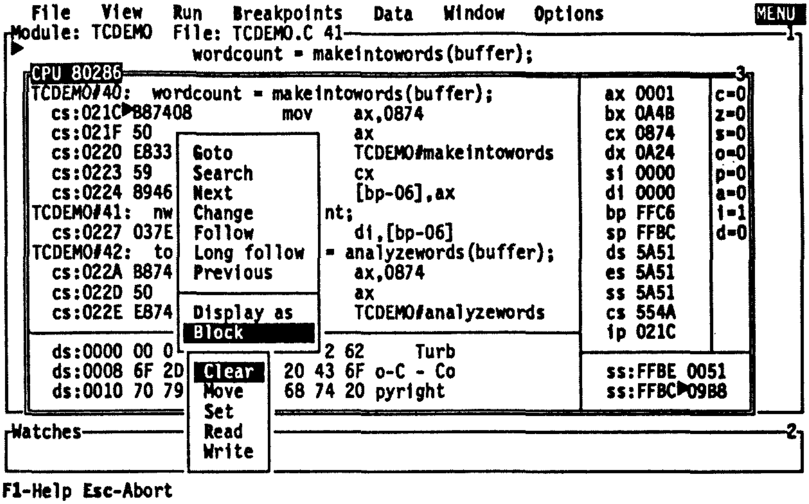
10バイト長の浮動小数点数を表示するようにセットします。浮動小数点数値が，科学的記数法で表示されます。
この表示形式は，80x87コプロセッサの内部表現形式です。Cのlong double型，Pascalのextended型（IEEE）に対応します。

Block

メモリのブロックを操作することができます。メモリブロックを移動，クリア，セットすることができます。またメモリブロックをディスクファイルに書き込んだり，ディスク

ファイルからメモリブロックに読み込んだりすることができます。

図10.8 Blockメニュー



Clear

メモリのブロックをゼロ (0) にセットします。先頭アドレスとバイト数の入力を求めます。

Move

メモリのブロックを，あるアドレスから別のアドレスにコピーします。コピー元のアドレス，コピー先のアドレス，およびコピーするバイト数の入力を求めます。

Set

メモリのブロックを指定されたバイト値にセットします。ブロックの先頭アドレス，セットするバイト数，セットする値の入力を求めます。

Read

ファイルの全体または一部を，メモリのブロックに読み込むことができます。読み込むファイルの名前，そのファイルを読み込むブロックの先頭アドレス，読み込むバイト数の入力を求めます。

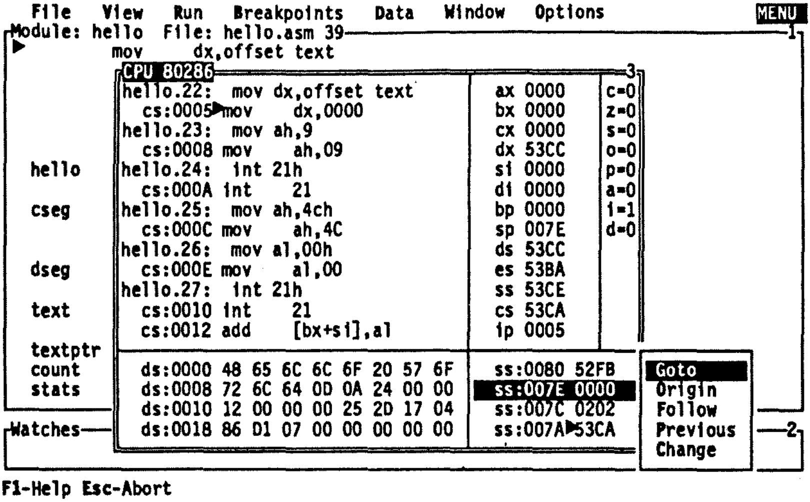
Write

メモリのブロックをファイルに書き込みます。書き込むファイル名，書き込むブロックの先頭アドレス，書き込むバイト数の入力を求めます。

Stackペインのローカルメニュー

GRPH-F10を押すと,Stackペインのローカルメニューがポップアップします。また,***CTRL***キーによるショートカットが有効になっているときは,***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのコマンドにアクセスすることもできます。

図10.9 Stackペインのローカルメニュー



Goto

Stackペインの表示を，スタック内の新しいアドレスに移動します。新しいスタックアドレスを入力してください。デバッグしているプログラムのスタックの外部のアドレスも，そうしたければ入力することができます。もっともスタックの外部のデータを調べるには，通常Dataペインを使います。アドレスの入力の仕方については，第9章を参照してください。

Previousコマンドは，Stackペインの表示を，Gotoコマンドを使用する前の位置に戻します。

Origin

Stackペインの表示を，SS:SPレジスタが示すカレントスタックロケーションに戻します。このコマンドは，最初の位置に戻りたいときに便利です。

Previousコマンドは，Stackペインの表示をOriginコマンドを使用する前の位置に戻します。

Follow

Stackペインの表示を,現在ハイライト表示されているワードにより指し示されるスタック内のワードに移動します。このコマンドは，スタックフレームを，呼び出し側の関数まで逆にたどるのに便利です。

Previousコマンドは，Stackペインの表示をFollowコマンドを使用する前の位置に戻します。

Previous

Stackペインの表示を，表示アドレスを変えるコマンドにより変える前の表示に戻します。矢印キーや***ROLLUP***および***ROLLDOWN***キーを使って表示アドレスを変えたときは，変更前の表示位置は記憶されません。

Previousを使うと，2つのアドレスに交互に切り替わります。

Change

現在ハイライト表示されているスタックワードに，新しいワード値を入力することができます。

このコマンドは，ハイライト表示されているスタック項目に，直接新しい値を入力することにより起動することもできます。入力を始めると，ローカルメニューでChangeコマンドを選択したときと同じプロンプトボックスが現われます。

アセンブラ

8086,80186および80286プロセッサと,8087,80287および80387数値演算コプロセッサの命令をアセンブルすることができます。

Turbo Debuggerの組み込みアセンブラを使ってプログラムを修正したときは,その変更は恒久的なものではありません。**Run/Program reset**コマンドを使ってプログラムを再ロードするか,または**File/Load**コマンドを使って別のプログラムをロードすると,そのプログラムに対して加えた変更は失われます。

アセンブラは,通常,プログラムを修正するためのアイデアを試してみるために使ってください。そしてその変更でうまくいくことを確かめたら,ソースコードを変更して,再コンパイルし,再リンクしてください。

オペランドのアドレスサイズのオーバーライド

組み込みアセンブラは,呼び出し命令(**CALL**),ジャンプ命令(**JMP**),条件ジャンプ命令(**JNE**,**JL**,その他)に対して,自動的に,目的アドレスに到達できる最も小さい機械語命令を生成します。特定のサイズでアセンブルしたいときは,目的アドレスの前に**NEAR**および**FAR**オーバーライドをつけて,サイズを指定することができます。たとえば次のようにします。

```
call    FAR  XYZ
jmp     NEAR A1
```

メモリオペランドと即値オペランド

プログラムの中で使用されているシンボルを命令のオペランドとして指定するときは,次の点に注意してください。

シンボルをそのまま指定したとき,および大カッコで囲んで指定したときは,そのシンボルの内容を指します。たとえば次のようにします。

```
mov     ax,[a]
mov     dx,a
```

シンボルのアドレスを指すときは，シンボルの前に**OFFSET**演算子を付けてください。

```
mov ax, offset a
```

オペランドのデータサイズのオーバーライド

オペランドの前に次の型指定をつけて，オペランドのサイズを指定しなければならない場合もあります。

```
BYTE PTR     (1バイト)
WORD PTR     (2バイト)
```

これらのオーバーライドを使った命令の例を示します。

```
add     BYTE  PTR [si],10
mov     word  PTR [bp+10], 99
```

8087/80287数値演算コプロセッサの命令をアセンブルするときは，これらのオーバーライドに加えて，次のオーバーライドを使うこともできます。

```
DWORD PTR     ( 4バイト)
QWORD PTR     ( 8バイト)
TBYTE PTR     (10バイト)
```

これらのオーバーライドを使った命令の例を示します。

```
fild    QWORD PTR [bx]
stp     TBYTE PTR [bp+4]
```

ストリング命令

ストリング命令で操作するデータのサイズ（バイトまたはワード）を指定するときは，オペランドに型指定をつけるのではなく，型指定のついた命令ニーモニックを使ってください。アセンブラは，サイズ指定のないニーモニックを使用して，オペランドにサイズ指定をつけるストリング命令の書式は受けつけません。**STOS WORD PTR ［DI］** ではなく，**STOSW**を使ってください。

Dumpウィンドウ

Dumpウィンドウは，メモリの領域のデータをそのまま表示します。このウィンドウの働きは，CPUウィンドウのDataペインとまったく同じです。

このウィンドウの表示内容とローカルメニューの説明は，この章の「**Dataペインのローカルメニュー**」の節を参照してください。

このウィンドウは，アセンブラのプログラムをソースレベルでデバッグしていて，データ領域のどこかをちょっと見てみたいときに使うことが最も多いでしょう。**View/Dump**コマンドを選択すると，Dumpウィンドウを開くことができます。

Inspectorウィンドウの中にいて，調べている対象を構成するバイトをそのまま見たい場合にも，このウィンドウを使うことができます。**View/Dump**コマンドを選択するとDumpウィンドウが開き，見たいデータが表示されます。

Registersウィンドウ

Registersウィンドウは，CPUのレジスタとフラグの内容を表示します。このウィンドウの働きは，CPUウィンドウのRegistersペインとFlagsペインの働きの組み合わせに似ています。

このウィンドウの表示内容とローカルメニューの説明は，この章の前部の「**Registersペインのローカルメニュー**」と「**Flagsペインのローカルメニュー**」の節を参照してください。

アセンブラプログラムをソースレベルでデバッグしていて，レジスタの値を見たいときは，このウィンドウを使ってください。Moduleウィンドウの大きさを縮小して，このウィンドウをその横に並べて表示することもできます。

Turbo Cのコード生成

Turbo Cコンパイラの，機械語コードを生成するときの動作は，複雑ですが予測可能です。そのためTurbo Cコンパイラに精通すると，機械語命令とソースコードの対応がすぐに正確にわかります。

関数の戻り値は，以下に示すレジスタに置かれます。

戻り値の型	レジスタ
int	AX
long	DX:AX
float	ST(0)
double	ST(0)
long double	ST(0)
near *	AX
far *	DX:AX

コンパイラはintおよびnearポインタに対しては頻繁にレジスタを使用します。このとき，最初はSIレジスタ，次にDIレジスタが用いられます。

自動変数および関数呼び出しでの引数は，SS:BPからアクセスされます。

AX，BX，CX，DXレジスタは，関数呼び出しの前後で保存/回復する必要はありません。

レジスタは常にワードレジスタとして使用され，**char**データ型についてもバイトレジスタとして使われることはありません。

switch文は，次の3つの形式のいずれか1つでコンパイルされます。これは，どれが最も効率的なコードが得られるかによって決められます。

■**if..else**の連鎖であるかのような条件ジャンプ

■コードアドレスのジャンプテーブル

■スイッチ値とコードアドレスのジャンプテーブル

第11章

80x87コプロセッサチップとエミュレータ

Turbo Debuggerには，浮動小数点数を扱うプログラムのデバッグのために，数値演算コプロセッサやソフトウェアエミュレータの状態を調べたり，変更したりする機能が用意されています。この章は，80x87数値演算コプロセッサの動作についてよく知っているプログラマのためのものです。浮動小数点数を扱うプログラムのデバッグでも，必らずしもこの章で説明されている機能を使う必要はありません。ただしこの機能を使うと，非常に発見が難しいバグを，ずっと容易に見つけることができる場合もあります。

この章では，まず，数値演算コプロセッサとソフトウェアエミュレータの違いについて説明します。次にNumeric Processorウィンドウの働きについて説明します。またその中で，浮動小数点レジスタ，ステータスビット，コントロールビットを調べたり，変更したりする方法を説明します。

80x87チップ対エミュレータ

Turbo Debuggerは，プログラムが数値演算チップまたはエミュレータのどちらかを使用するときは，自動的にそれを検出し，どちらが使用されているかにしたがって自分の動作を合せます。

ほとんどのプログラムは，エミュレータまたは数値演算チップのどちらか1つを使用し，同一プログラム内で両方を使用しないことに注意してください。両方を使用する特別のアセンブラコードを書いたときは，Turbo Debuggerは，数値演算チップの状態を表示することができません。エミュレータの状態だけを表示します。

Numeric Processorウィンドウ

メインメニューバーからView/Numeric processorコマンドを選択すると，Numeric Processorウィンドウが開きます。ウィンドウのタイトル行には，最後に実行された浮動小数点演算に関する情報が表示されます。命令ポインタとデータポインタの値，それに命令のオペコードです。命令ポインタとデータポインタは，どちらも20ビットの物理アドレスで示されます。上位の4桁をセグメント値として，下位の1桁をオフセット値として使うと，こららのアドレスを，セグメント：オフセット形式に変換することができます。

たとえば，最上行にIPTR=5A669と表示されている場合に，CPUウィンドウで命令ポインタの指し示す命令を調べたいときは，アドレス5a66:9を表示させれば，その命令を見ることができます。このウィンドウには，3つのペインがあります。左側のペイン(Registerペイン)には，浮動小数点レジスタの内容が表示されます。中央のペイン(Controlペイン)には，コントロールフラグが表示されます。右側のペイン(Statusペイン)にはステータスフラグが表示されます。

図11.1　Numeric Processorウィンドウ

```
  File  View  Run  Breakpoints  Data  Window  Options                     READY
┌Module: TPDEMO  File: TPDEMO.PAS 74──────────────────────────────────────1┐
│    AvgWords := NumWords / NumLines
│  else
│    AvgWords := 0;
│  Writeln;     ┌Emulator IPTR=00000 OPCODE=000 OPTR=00000──────────4┐
│  Writeln(Nu│Valid ST(0) 41                │ im=0  │ ie=0 │
│          Nu│Valid ST(1) 5                 │ dm=0  │ de=0 │
│          Nu│Empty ST(2)                   │ zm=0  │ ze=0 │
│  Writeln('A│Empty ST(3)                   │ om=0  │ oe=0 │
│  Writeln;   │Empty ST(4)                   │ um=1  │ ue=0 │
│             │Empty ST(5)                   │ pm=1  │ pe=0 │
│  { Dump wor│Empty ST(6)                   │iem=0  │ ir=0 │
│  Write('Wor│Empty ST(7)                   │ pc=3  │ cc=9 │
│  for i := 1│                              │ rc=0  │ st=2 │
│    Write(i:│                              │ ic=1  │      │
│  Writeln;   └──────────────────────────────────────────────┘
│
│  Write('Frequency:  ');
│  for i := 1 to MaxWordLen do
└──────────────────────────────────────────────────────────────────────────┘
┌Watches─────────────────────────────────────────────────────────────────2┐
│
└──────────────────────────────────────────────────────────────────────────┘
F2-Bkpt F3-Close F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

タイトル行のIPTRは，最後の浮動小数点命令がフェッチされたアドレスを20ビットの物理アドレスで示します。OPCODEは，フェッチされた命令のバイトを示します。OPTR

は，その命令が参照したメモリのアドレス(もし参照していれば)を20ビットの物理アドレスで示します。

80ビット浮動小数点レジスタ

Registerペインは，浮動小数点レジスタST(0)～ST(7)の状態(Valid/Zero/Sepcial/Empty)と，その内容を表示します。レジスタの内容は，80ビットの浮動小数点数として表示されます。

Numeric Processorウィンドウがズームされているか(F5)，幅が拡大されている(Window/Move/Resize)ときは，浮動小数点レジスタが16進数バイトでも表示されます。

ステータスビット

次の表は，各ステータスフラグの内容と，Statusペインに表示される略号を示します。

表示	フラグの説明
ie	無効演算
de	非正規化オペランド
ze	ゼロによる割算
oe	オーバーフロー
ue	アンダーフロー
pe	精度
ir	割り込み要求
cc	コンディションコード
st	スタックトップポインタ

コントロールビット

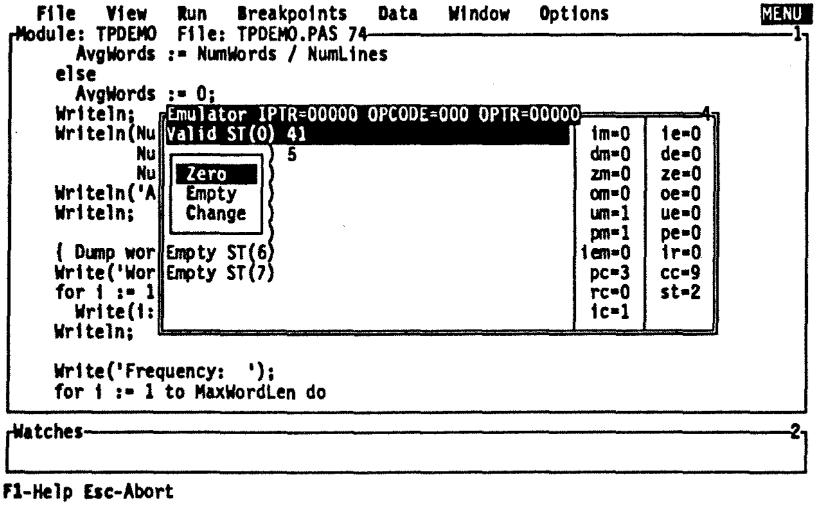
次の表は，各コントロールフラグの内容と，Controlペインに表示される略号を示します。

表示	フラグの説明
im	無効演算マスク
dm	非正規化オランドマスク
zm	ゼロによる割算マスク
om	オーバーフローマスク
um	アンダーフローマスク
pm	精度マスク
iem	割り込み有効マスク(8087のみ)
pc	精度コントロール
rc	丸めコントロール
ic	無限大コントロール

Registerペインのローカルメニュー

GRPH-F10を押すと，***Register***ペインのローカルメニューがポップアップします。また***CTRL***によるショートカットが有効になっているときは，***CTRL***とローカルコマンドの先頭文字を一緒に押すことにより，そのコマンドに直接アクセスすることができます。

図11.2　Registerペインのローカルメニュー



Zero

現在ハイライト表示されているレジスタをゼロにセットします。

Empty

現在ハイライト表示されているレジスタの値を〝空〟にセットします。空状態は，レジスタに有効なデータが含まれていないことを示す特別な状態です。

Change

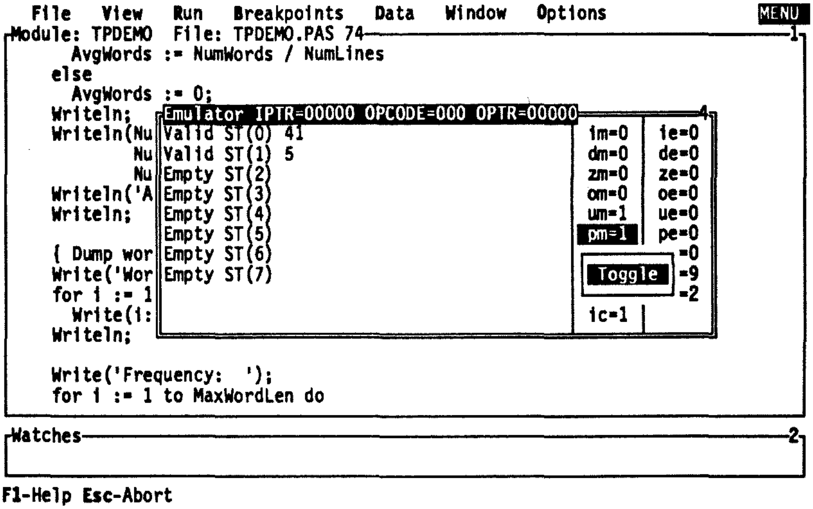
現在ハイライト表示されているレジスタに新しい値をロードします。ロードする値の入力を求めます。Cの式要素のすべてを使って，整数または浮動小数点数を入力することができます。入力した値は自動的に，数値プロセッサにより使用される80ビットの実数形式に変換されます。

このコマンドは，浮動小数点レジスタにロードしたい新しい値を直接入力することにより，行使することもできます。値の入力を始めると，Changeコマンドを選択したときと同じプロンプトボックスが現われます。

Statusペインのローカルメニュー

TABを押してStatusペインに移り，***GRPH-F10***を押すと，Statusペインのローカルメニューがポップアップします。また***CTRL***キーと希望するコマンドの先頭文字を一緒に押すと，そのコマンドに直接アクセスすることができます。

図11.3　Statusペインのローカルメニュー



Toggle

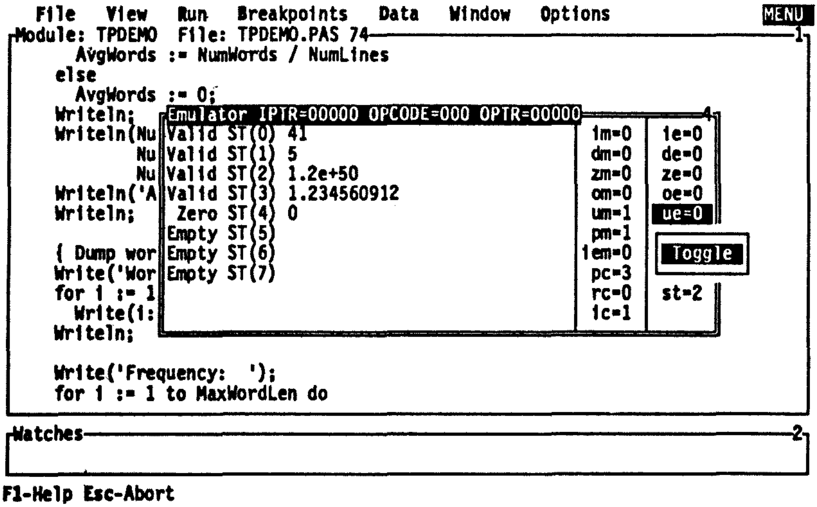
現在ハイライト表示されているステータスフラグをセットする値を交互に切り替えます。ほとんどのフラグは，セットとクリア(1または0)の2つの状態しかとりません。それらのフラグに対しては，フラグを現在の値からもう1つの値にトグルします。その他のフラグには3つ以上の値をとるものもあります。そのようなフラグに対しては，フラグの値を最大値に達するまでインクリメントし，最大値に達したらゼロに戻します。

ステータスフラグの値は，**リターン**を押すことによりトグルすることもできます。

Controlペインのローカルメニュー

SHIFT-TABを押してControlペインに移り，***GRPH-F10***を押すと，Controlペインのローカルメニューがポップアップします。また***CTRL***キーと希望するコマンドの先頭文字を一緒に押すと，そのコマンドに直接アクセスすることができます。

図11.4　Controlペインのローカルメニュー



Toggle

現在ハイライト表示されているステータスフラグをセットする値を交互に切り替えます。ほとんどのフラグは，セットとクリア(1または0)の2つの状態しかとりません。それらのフラグに対しては，フラグを現在の値からもう1つの値にトグルします。その他のフラグには3つ以上の値をとるものもあります。そのようなフラグに対しては，フラグの値を最大値に達するまでインクリメントし，最大値に達したらゼロに戻します。

ステータスフラグの値は，**リターン**を押すことによりトグルすることもできます。

第12章

コマンドリファレンス

すべてのコマンドの説明が終わったので，この章には一覧表を示します。この章には，以下のコマンドがまとめてあります。

■ファンクションキーやその他のキーで入力することができる，すべてのシングルキーストロークコマンド

■メインメニューコマンドと各ウィンドウのすべてのローカルメニューコマンド

■2種類のペインにおいて使用されるキーストローク，テキストの入力を求めるプロンプトに対する入力，新しいウィンドウサイズと位置を求めるプロンプトに対する入力

ホットキー

ホットキーは，Turbo Debuggerの環境の中であればどこからでも使えるキーです。次の表12.1に，すべてのホットキーを掲載します。

表12.1 ホットキーコマンド

キー	メニューコマンド	機能
F1		現在の状況に応じたヘルプを表示する。
F2	Breakpoints/Toggle	カーソルの位置にブレークポイントを設定する。
F3	Window/Close	カレントウィンドウを閉じる。
F4	Run/Go to cursor	カーソル位置まで実行する。
F5		カレントウィンドウをズーム/アンズームする。
F6	Window/Next	カレントウィンドウを切り替える。
F7	Run/Trace into	1ソース行または1命令実行する。
F8	Run/Step over	1ソース行または1命令実行する。呼び出しをスキップする。
F9	Run/Run	プログラムを実行する。
F10		メインメニューバーに戻る，メニューから抜け出す。
GRPH-F1		最後に見たヘルプ画面をもう一度表示する。
GRPH-F2	Breakpoints/At	指定されたアドレスにブレークポイントを設定する。
GRPH-F3	View/Module	モジュールピックリストを表示する。
GRPH-F4	Run/Animate	連続的に表示を更新しながらプログラムを実行する。
GRPH-F5	View/User screen	プログラムの画面を表示する。
GRPH-F6	Window/Undo close	最後に閉じたウィンドウをもう一度開く。
GRPH-F7	Run/Instruction trace	1命令実行する。
GRPH-F8	Run/Until return	呼び出し側に戻るまで実行する。
GRPH-F9	Run/Run to	指定されたアドレスまで実行する。
GRPH-F10		ローカルメニューを呼び出す。
GRPH-0	Window/Window pick	現在開いているウィンドウのリストを表示する。
GRPH-1～9		その番号のウィンドウをアクティブにする。

GRPH-F		Fileメニューに入る。
GRPH-V		Viewメニューに入る。
GRPH-R		Runメニューに入る。
GRPH-B		Breakpointsメニューに入る。
GRPH-D		Dataメニューに入る。
GRPH-W		Windowメニューに入る。
GRPH-O		Optionsメニューに入る。
GRPH-X		Turbo Debuggerを終了してDOSに戻る。
GRPH-^ (***GRPH-=***)	Options/Macros/Create	キーストロークマクロを設定する。
GRPH--	Options/Macros/Stop recording	マクロの記録を終了する。
CTRL-F2	Run/Program reset	デバッグセッションを停止し，再開のためにプログラムを再ロードする。
CTRL-F4	Data/Evaluate	式を評価する。
CTRL-F7	Data/Watch	Watchesウィンドウに変数を追加する。
CTRL-F8	Breakpoints/Toggle	カーソル位置にブレークポイントを設定，解除する。
CTRL-F9	Run/Run	プログラムを実行する。
CTRL-F10		ローカルメニューを呼び出す。
CTRL-右矢印		CPUウィンドウのCode,Data,Stackペインの表示開始アドレスを1バイト高位にずらす。
CTRL-左矢印		CPUウィンドウのCode,Data,Stackペインの表示開始アドレスを1バイト下位にずらす。
ESC		Inspectorウィンドウを閉じる，メニューから抜け出す。
INS		テキストブロックの指定（ハイライト表示）を開始する。範囲の指定には**左右の矢印キー**を使う。
GRPH-XFER (***ScrollLock***)	Window/Move/Resize	ウィンドウの位置と形状を変更する。
TAB		次のウィンドウペインに移る。
SHIFT-TAB		前のウィンドウペインに戻る。
SHIFT-矢印		ウィンドウペイン間の移動，矢印の方向のペインがアクティブになる。

メインメニューバーから選択できるコマンド

F10を押すと，Turbo Debuggerのどこからでも，メインメニューに戻ることができます。メインメニューの中のメニュータイトルの上にカーソルを移動して**リターン**を押すか，メニュータイトルの先頭文字を押すことにより，メインメニューからその下の各メニューに入ることができます。また**GRPH**と希望するメニューの先頭文字を一緒に押すと，そのメニューを直接（メインメニューに戻らないで）開くこともできます。

Fileメニュー

Load	デバッグする新しいプログラムをロードする
Change dir	ディスクドライブまたはディレクトリを変更する
Get info	デバッグセッションの現在の状態に関する情報を表示する
OS shell	DOSのコマンドプロセッサを起動する
Quit	Turbo Debuggerを終了してDOSに戻る

Viewメニュー

Breakpoints	Breakpointsウィンドウを開く
Stack	Stackウィンドウを開く
Log	Logウィンドウを開く
Watches	Watchesウィンドウを開く
Variables	Variablesウィンドウを開く
Module	Moduleウィンドウを開く
File	Fileウィンドウを開く
CPU	CPUウィンドウを開く
Dump	Dumpウィンドウを開く
Registers	Registersウィンドウを開く
Numeic processor	Numeric Processorウィンドウを開く
User screen	デバッグしているプログラムの画面を表示する

Another	
Module	新しいModuleウィンドウを開く
Dump	新しいDataウィンドウを開く
File	新しいFileウィンドウを開く

Runメニュー

Run	プログラムを無制約に実行する
Program reset	同じプログラムを再ロードする
Go to cursor	現在のカーソルの位置まで実行する
Trace into	1ソース行または1命令実行する
Step over	呼び出しをスキップしながらトレースする
Execute to	指定されたアドレスまで実行する
Until return	呼び出し側に戻るまで実行する
Animate	1ソース行または1命令ずつ表示を更新しながら実行する
Instruction trace	1命令実行する

Breakpointsメニュー

Toggle	カーソル位置にブレークポイントを設定，解除する
At	指定されたアドレスにブレークポイントを設定する
Changed memory global	指定されたメモリ領域の値が変化したときトリガされるグローバルなブレークポイントを設定する
Expression true global	式が真のときトリガされるグローバルなブレークポイントを設定する
Delete all	すべてのブレークポイントを解除する

Dataメニュー

Inspect	データを表示する
Evaluate/modify	式を評価する
Watch	Watchウィンドウに変数を追加する

Function return	ルーチンの呼び出しと戻り値に関する情報を表示する
Function return	ルーチンの呼び出しと戻り値に関する情報を表示する

Windowメニュー

Window pick	開いているウィンドウのリストからウィンドウを選択する
Next pane	カレントウィンドウ内の次のペインに移る
Move/resize	カレントウィンドウの位置と形状を変更する
Close	カレントウィンドウを閉じる
Undo close	最後に閉じたウィンドウをもう一度開く
Dump pane to log	カレントペインをLogウィンドウに書き込む
Restor standard	デフォルトのウィンドウ配置に戻す
Screen repain	画面全体を再表示する

Optionsメニュー

Language	
Source module	カレントモジュールのソースファイルの名前から式を評価する言語を決定する
C	式の評価にCを使う
Pascal	式の評価にPascalを使う
Assember	式の評価にアセンブラを使う
Macros	
Create	キーストロークマクロを設定する
Stop recording	記録セッションを終了する
Remove	キーストロークマクロを解除する
Delete all	すべてのキーストロークマクロを解除する
Environment	
Interger format	Hex/Decimal/Both(数の表現形式)
Display swapping	None/Smart/Always(ユーザ画面の退避モード)
Screen size	デバッガの画面サイズを設定する PC-9801：Standard 25 line/High density 46 line IBM PC：25 line / 43/50 line

Tab size	テキストファイルを表示しているときのタブ間隔
Path for source	ソースファイルを探すディレクトリのリスト
Arguments	プログラムのコマンドライン引数を設定する
Save options	オプション，マクロ，ウィンドウ配置をコンフィギュレーションファイルに書き込む
Restore options	ディスクからオプションの設定を読み出す

Hardware status（PC-9801のみ）

Visual user G-page	表示するGVRAMのバンク(0/1)を指定する
Active user G-page	デバッガからアクセスするGVRAMのバンク(0/1)を指定する
Graphics switch	ユーザグラフィックス画面の表示/非表示を指定する
Cursor switch	DOS/On/Off（カーソルの扱い）
EMS/VRAM switch	EMS/VRAM（デバッガからアクセスするバンク）

Localメニューコマンド

CTRL-F10を押すことにより，カレントウィンドウのポップアップメニューすなわち〝ローカル〟メニューを呼び出すことができます。***CTRL***キーによるショートカットが有効になっているときは，***CTRL***キーと希望するコマンドの先頭文字を一緒に押すことにより，そのコマンドを直接起動することができます(***CTRL***キーによるショートカットは，インストールプログラムTDINSTを使って有効にすることができます)。

ローカルメニューは，ウィンドウの種類(*Breakpoints*, *Module*, その他)とウィンドウ内のペインにより異なります。次に，各ウィンドウおよびウィンドウ内の各ペインのローカルメニューコマンドの要約を示します。

いくつかのペインには，そのローカルメニューコマンドの中で最もよく使用されるコマンドに対するショートカットが用意されています。これらのショートカットがあるときは，そのウィンドウまたはペインのローカルメニューコマンドのリストの後に説明してあります。多くのペインでは，**リターン**キーが現在ハイライト表示されている項目を調べたり，変更したりするコマンドのショートカットになっています。また***DEL***キーが，ハイライト表示されている項目を削除するコマンドへのショートカットになっているペインもあります。さらにいくつかのペインでは，必要なローカルメニューコマンドを先に選択することなく直接文字や数字を入力すると，そのローカルメニューコマンドのプロンプトボックスが現われて，入力した文字や数字がその中に入ります。

Breakpointsウィンドウのローカルメニュー

Breakpointsウィンドウには，Listペイン(左側)とDetailペイン(右側)の2つのペインがあります。ローカルメニューがあるのは，Listペインだけです。

Set action	
Break	プログラムの実行を停止するブレークポイントを設定する
Log	式の値をログに記録するブレークポイントを設定する
Execute	式を実行するブレークポイントを設定する
Conditon	
Always	無条件
Changed memory	メモリの値が変化したとき

Expression true	式が真のとき
Hardware	
Cycle type	
Read memory	メモリ読み出しにマッチ
Write memory	メモリ書き込みにマッチ
Access memory	メモリ読み出しおよび書き込みにマッチ
Input I/O	I/O読み込みにマッチ
Output I/O	I/O書き込みにマッチ
Both I/O	I/O読み込みおよび書き込みにマッチ
Fetch instruction	命令フェッチにマッチ
Address	
Above	指定されたアドレスより大きいときマッチ
Below	指定されたアドレスより小さいときマッチ
Range	指定されたアドレス領域内のときマッチ
Not range	指定されたアドレス領域外のときマッチ
Less or equal	指定されたアドレス以下のときマッチ
Greater or equal	指定されたアドレス以上のときマッチ
Equal	指定されたアドレスに等しいときマッチ
Unequal	指定されたアドレス以外のときマッチ
Match all	すべてのアドレスにマッチ
Data	
Above	指定された値より大きいときマッチ
Below	指定された値より小さいときマッチ
Range	指定された値の範囲内のときマッチ
Not range	指定された値の範囲外のときマッチ
Less or equal	指定された値以下のときマッチ
Greater or equal	指定された値以上のときマッチ
Equal	指定された値に等しいときマッチ
Unequal	指定された値以外のときマッチ
Match all	すべての値にマッチ
Pass count	ブレークポイントをスキップする回数
Enable/disable	ブレークポイントを有効または無効にする
Add	新しいブレークポイントを追加する
Global	新しいグローバルブレークポイントを追加する
Remove	ハイライト表示されているブレークポイントを解除する

Delete all	すべてのブレークポイントを解除する
Inspect	ブレークポイントが設定されているコードを表示する

ハイライトバーをブレークポイントに置いて**リターン**を押すと，Inspectコマンドへのショートカットになります。

CPUウィンドウのローカルメニュー

CPUウィンドウには，Codeペイン，Dataペイン，Stackペイン，Registerペイン，Flagsペインの5つのペインがあります。各ペインにはそれぞれローカルメニューがあります。

Codeペインのローカルメニュー

Goto	新しいアドレスのコードを表示する
Origin	CS:IPのコードを表示する
Follow	JMPまたはCALLの目的アドレスのコードを表示する
Caller	現在のルーチンを呼び出したコードを表示する
Previous	表示アドレスを変更した最後のコマンドにより変更する前のアドレスのコードを表示する
Search	命令またはバイトを表示する
View source	Moduleウィンドウに切り替える
Mixed	No/Yes/Both (逆アセンブルコードにソースコードを組み合わせて表示する)
New CS:IP	CS:IPを新しいアドレスにセットする
Assemble	カーソルの置かれている命令をアセンブルする
I/O	
In byte	I/Oポート（ロケーション）から1バイト読み込む
Out byte	I/Oポート（ロケーション）に1バイト書き込む
Read word	I/Oポート（ロケーション）から1ワード読み込む
Write word	I/Oポート（ロケーション）に1ワード書き込む

Dataペインのローカルメニュー

Goto	新しいアドレスのデータを表示する
Search	文字列またはデータバイトを検索する
Next	検索を続行し次の出現を探す
Change	カーソルのあるアドレスのデータバイトを変更する
Follow near	ポインタの指し示す先をたどる
Long follow	farポインタの指し示す先をたどる
Previous	表示アドレスを変更した最後のコマンドにより変更する前のアドレスのデータを表示する
Display as	
Byte	バイトを16進数で表示する
Word	ワードを16進数で表示する
Long	ロングワード（4バイト）を16進数で表示する
Comp	Pascalのcomp型整数（8バイト）を表示する
Float	ショート型（4バイト）浮動小数点数（Pascalのsingle型）を表示する
Real	6バイトの浮動小数点数（Pascalのreal型）を表示する
Double	8バイトの浮動小数点数を表示する
Extended	10バイトの浮動小数点数（Cのlong double型）を表示する
Block	
Clear	メモリのブロックをゼロにセットする
Move	メモリのブロックを移動する
Set	メモリのブロックに値をセットする
Read	ファイルからメモリに読み込む
Write	メモリをファイルに書き込む

値を変更したいアドレスにカーソルを置いて新しい値を入力すると, Changeコマンドへのショートカットになります。

Stackペインのローカルメニュー

Goto	スタック内の新しいアドレスのデータを表示する
Origin	SS:SPのデータを表示する

Follow	ハイライト表示されているデータにより指し示されるデータを表示する
Previous	表示アドレスを変更した最後のコマンドによる変更前の表示に戻す
Change	ハイライト表示されているデータの値を変更する

値を変更したいデータにハイライトバーを置いて新しい値を入力すると，**Change**コマンドへのショートカットになります。

Registerペインのローカルメニュー

Increment	ハイライト表示されているレジスタに1加える
Decrement	ハイライト表示されているレジスタから1を引く
Zero	ハイライト表示されているレジスタをクリアする
Change	ハイライト表示されているレジスタに新しい値をセットする
Registers 32-bit	No/Yes（レジスタの16ビット/32ビット表示の選択）

値を変更したいレジスタにハイライトバーを置いて新しい値を入力すると，**Change**コマンドへのショートカットになります。

Flagsペインのローカルメニュー

Toggle	ハイライト表示されているフラグをセットまたはクリアする

値を変更したいフラグにハイライトバーを置いて**リターン**を押すと，**Toggle**コマンドのショートカットになります。

Fileウィンドウのローカルメニュー

Fileウィンドウは，ディスクファイルの内容を表示します。

Goto	指定された行またはオフセットを表示する
Search	文字列またはバイトリストを検索する
Next	検索を続行して次の出現を探す
Display as	Ascii/Hex：ファイルの表示形式を設定する
File	新しいファイルを表示する

Edit	テキストファイルの編集または16進数バイトの変更を可能にする

文字列またはバイトリストの入力は，Searchコマンドのショートカットになります。

Logウィンドウメニューのローカルメニュー

Logウィンドウは，ログに送られた情報を表示します。

Open log file	ディスクファイルへのログを開始する
Close log file	ディスクファイルへのログを停止する
Logging	No/Yes：ログを取るかどうかを選択する
Add comment	ログにコメントを書き込む
Erase log	ログを消去する

コメントの入力は，Add commentコマンドへのショートカットになります。

Moduleウィンドウのローカルメニュー

Moduleウィンドウは，プログラムモジュールのソースファイルを表示します。

Inspect	カーソルの置かれている変数の内容を表示する
Watch	カーソルの置かれている変数をウオッチ リストに追加する
Module	別のモジュールを表示する
File	別のソースファイルを表示する
Previous	カレントモジュールの前に見ていたモジュールの最後に見ていた場所，表示移動コマンドを発行する前に見ていたカレントモジュールの場所を表示する
Line	モジュールの指定された行を表示する
Search	文字列を検索する
Next	検索を続行し，次の出現を探す
Origin	カレントプログラムロケーションを表示する
Goto	指定されたアドレスを表示する
Edit	エディタを起動してソースファイルの編集を可能にする

表示させたいアドレス(行番号，関数名，16進数アドレス)への入力は，Gotoコマンドへのショートカットになります。

Numeric Processorウィンドウのローカルメニュー

Numeric Processorウィンドウには，Registerペイン，Statusペイン，Controlペインの3つのペインがあります。各ペインには，それぞれローカルメニューがあります。

Registerペインのローカルメニュー

Zero	ハイライト表示されているレジスタをゼロにセットする
Empty	ハイライト表示されているレジスタを空にする
Change	ハイライト表示されているレジスタに値をセットする

レジスタにセットしたい値を入力すると，Changeコマンドへのショートカットになります。

Statusペインのローカルメニュー

Toggle	ハイライト表示されているステータスフラグをセットできる値を循環的にセットする

リターンキーはToggleコマンドへのショートカットになります。**リターン**キーを押すと，ハイライト表示されているフラグの値が循環的に変化します。

Controlペインのローカルメニュー

Toggle	ハイライト表示されているコントロールフラグに，そのフラグにセットできる値を循環的にセットする

リターンキーはToggleコマンドへのショートカットです。ハイライト表示されているフラグの値が循環的に変化します。

Stackウィンドウのローカルメニュー

Stackウィンドウは，現在アクティブなすべての関数を表示します。

Inspect	ハイライト表示されている関数のソースコードを表示する
Locals	ハイライト表示されている関数にローカルな変数を表示する

リターンキーを押すとInspectコマンドへのショートカットになります。

Variableウィンドウのローカルメニュー

Variableウィンドウには，Global SymbolペインとLocal Symbolペインの2つのペインがあります。各ペインには，それぞれローカルメニューがあります。

Global Symbolペインのローカルメニュー

Inspect	ハイライト表示されているシンボルの内容を表示する
Change	ハイライト表示されているシンボルの値を変更する

リターンキーを押すとInspectコマンドへのショートカットになります。

Static Symbolペインのローカルメニュー

Inspect	ハイライト表示されているシンボルの内容を表示する
Change	ハイライト表示されているシンボルの値を変更する

リターンキーを押すとInspectコマンドへのショートカットになります。

Watchesウィンドウのローカルメニュー

Watches ウィンドウは，ウオッチしている変数の名前と現在の値を表示します。

Watch	ウォッチする変数を追加する
Edit	ウィンドウ内での編集を可能にする
Remove	ハイライト表示されている変数を削除する
Delete all	ウィンドウの中のすべての変数を削除する
Inspect	ハイライト表示されている変数の内容を表示する
Change	ハイライト表示さる変数の値を変更する

リターンキーを押すとEditコマンドへのショートカットになります。

Inspectorウィンドウのローカルメニュー

Inspectorウィンドウは，データの内容を表示します。

Range	表示する配列の要素を選択する
Change	ハイライト表示されている項の値を変更する
Inspect	新しいInspectorウィンドウを開いて，ハイライト表示されている項の内容を表示する
Descend	現在のInspectorウィンドウの中に，ハイライト表示されている項の内容を表示する
New expression	現在のInspectorウィンドウの中に，指定された変数の内容を表示する

Textペイン

ここでは，テキストファイルの内容を表示するすべてのペインを総称してTextペインと呼びます。Textペインでは，点滅するカーソルが，ファイル内のカレントポジションを示します。Textペインには，次のようなローカルメニューコマンドがあります。

キー	機能
INS	テキストブロックの範囲を指定する
↑	カーソルを1行上に移動する
↓	カーソルを1行下に移動する
→	カーソルを1桁右に移動する
←	カーソルを1桁左に移動する
CTRL-→	カーソルを次の語に移動する
CTRL-←	カーソルを前の語に移動する
HOME	カーソルを行の先頭に移動する
SHIFT-HOME(End)	カーソルを行の最後の文字に移動する
ROLLDOWN (PgUp)	1画面分上にスクロールする
ROLLUP (PgDn)	1画面分下にスクロールする
CTRL-HOME	カーソルをペインの最上行に移動する
CTRL-SHIFT-HOME (Ctrl-End)	カーソルをペインの最下行に移動する
CTRL-ROLLDOWN (Ctrl-PgUp)	カーソルをファイルの先頭行に移動する
CTRL-ROLLUP (Ctrl-PgDn)	カーソルをファイルの最後の行に移動する

CTRLキーによるショートカットを使っていないときは，カーソル移動とスクロールのために，WordStar方式の***CTRL***キーと文字キーの組み合わせを使うことができます。

キー	機能
CTRL-S	カーソルを1桁右に移動する
CTRL-D	カーソルを1桁左に移動する
CTRL-E	カーソルを1行上に移動する
CTRL-X	カーソルを1行下に移動する
CTRL-R	1画面分上にスクロールする
CTRL-C	1画面分下にスクロールする
CTRL-F	カーソルを次の語に移動する
CTRL-A	カーソルを前の語に移動する

Listペイン

ここでは，リストを表示するすべてのペインを総称して，Listペインと呼びます。Listペインでは，ハイライトバーが，リスト内のカレントポジションを示します。Listペインには，次のようなローカルメニューコマンドがあります。

キー	機能
↑	ハイライトバーを1項目上に移動する
↓	ハイライトバーを1項目下に移動する
HOME	ハイライトバーを行の先頭に移動する
SHIFT-HOME(End)	ハイライトバーを行の最後に移動する
ROLLDOWN (PgUp)	1画面分上にスクロールする
ROLLUP (PgDn)	1画面分下にスクロールする
CTRL-HOME	ハイライトバーをペインの最上行に移動する
CTRL-SHIFT-HOME (Ctrl-End)	ハイライトバーをペインの最下行に移動する
CTRL-ROLLDOWN	ハイライトバーをリストの先頭の項目に移動する
CTRL-ROLLUP	ハイライトバーをリストの最後の項目に移動する
BS(Backspace)	項目検索文字列から1文字削除する
文字	項目検索文字列を入力する

CTRLキーによるショートカットを使っていないときは，カーソル移動とスクロールのために，WordStar方式の***CTRL***キーと文字キーの組み合わせを使うことができます。

キー	機能
CTRL-E	ハイライトバーを1行上に移動する
CTRL-X	ハイライトバーを1行下に移動する
CTRL-R	1画面分上にスクロールする
CTRL-C	1画面分下にスクロールする

プロンプトボックスの中にいるときのコマンド

プロンプトボックスの中にいるときは，次のコマンドが使用できます。

キー	機能
↑	カーソルを1ヒストリ項目上に移動する
↓	カーソルを1ヒストリ項目下に移動する
→	カーソルを1文字右に移動する
←	カーソルを1文字左に移動する
CTRL-→	カーソルを次の語に移動する
CTRL-←	カーソルを前の語に移動する
HOME	カーソルを行の先頭の文字に移動する
SHIFT-HOME(End)	カーソルを行の最後の文字に移動する
ROLLDOWN	1画面分上にスクロールする
ROLLUP	1画面分下にスクロールする
CTRL-HOME	カーソルをペインの最上行に移動する
CTRL-SHIFT-HOME (Ctrl-End)	カーソルをペインの最下行に移動する
CTRL-ROLLDOWN	カーソルをリストの先頭の項目に移動する
CTRL-ROLLUP	カーソルをリストの最後の項目に移動する
BS(Backspace)	カーソルの前の文字を削除する
リターン	入力を終了する
DEL	カーソルの位置の文字を削除する
ESC	プロンプトをキャンセルしてメニューに戻る

ウィンドウ操作コマンド

キー	機能
GRPH-XFER (ScrollLock)	ウィンドウ位置決めモードをオン，オフする
↑	ウィンドウを1行上に移動する
↓	ウィンドウを1行下に移動する
→	ウィンドウを1桁右に移動する
←	ウィンドウを1桁左に移動する
SHIFT- ↑	ウィンドウの下辺を上に移動する
SHIFT- ↓	ウィンドウの下辺を下に移動する
SHIFT-←	ウィンドウの右辺を左に移動する
SHIFT-→	ウィンドウを右辺を右に移動する
HOME	画面の左端に移動する
SHIFT-HOME(End)	画面の右端に移動する
ROLLDOWN (PgUp)	画面の最上行に移動する
ROLLUP (PgDn)	画面の最下行に移動する
リターン	現在の位置に設定する
ESC	ウィンドウ位置決めコマンドをキャンセルする

検索のワイルドカード

次の2つの場合に，ワイルドカードを使うことができます。

■プログラムをロードするため，またはファイルの内容を調べるために，ファイル名を入力するとき
■Textペインの中で検索文字列を入力するとき

疑問符(?)は，1文字に対するワイルドカードで，すべての文字にマッチします。アスタリスク(*)は，文字列に対するワイルドカードで，ヌル文字列（長さゼロの文字列）を含むすべての文字列にマッチします。

ファイルリスト

ファイル名の入力が求められているときに次のどれかを入力すると，ファイルおよびディレクトリのリストが表示されます。

■ワイルドカード文字(＊または?)を含むファイル名
■ディスクドライブ文字，たとえばA:
■ディレクトリ名，たとえば¥MYDIR

ファイルのリストとディレクトリのリストは，2つのListペインに別々に表示されます。左側のペインには，指定したディレクトリの中のファイルのリストが表示されます。右側のペインには，指定したディレクトリの親ディレクトリ(..¥)と，そのディレクトリのサブディレクトリのリストが表示されます。複数のペインのあるウィンドウの場合と同じように，***TAB***および***SHIFT-TAB***を押すことにより，1方のペインから他方のペインに移ることができます。

ハイライトバーを右側のペインのディレクトリのどれかに置いて**リターン**を押すと，そのディレクトリに移ることができます。新しいディレクトリに移ると，そのディレクトリの中のマッチするファイルが表示されます。

別のディレクトリを探したいとき，または異なるワイルドカードマスクで探したいときは，***INS***を押すと，ファイル名(ワイルドカードを含む)およびカレントディレクトリを編集することができます。

またファイルリストでは，他のすべてのリストと同じように，希望するファイル名の入力を開始することにより，そのファイル名に到達することができます。

ハイライトバーを希望するファイル名に移動して**リターン**を押すと，そのファイルを選択することができます。

メニュー構成

Turbo Debuggerのメニュー(メインメニューとその下のプルダウンメニュー)の構造を,**図12.1**, **図12.2**, **図12.3**に示します。

図12.1　File, View, Runメニュー

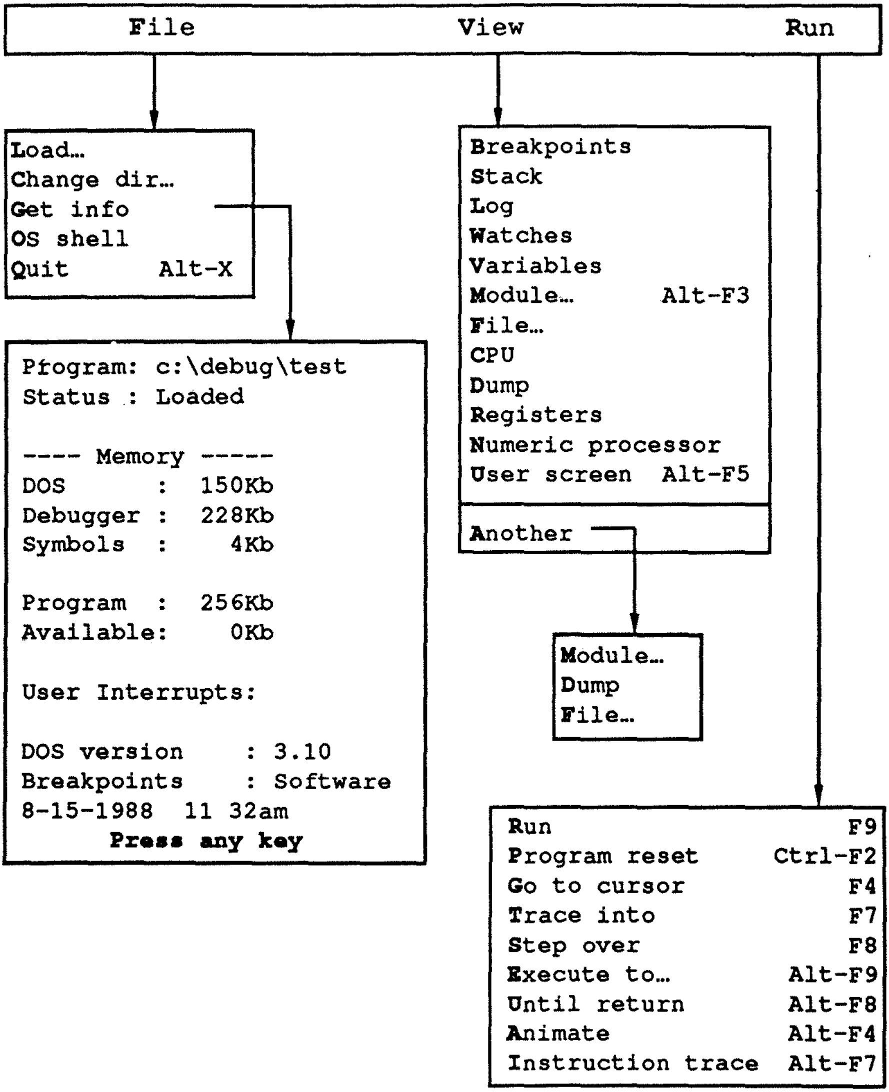


図12.2　Breakpoints, Data, Windowメニュー

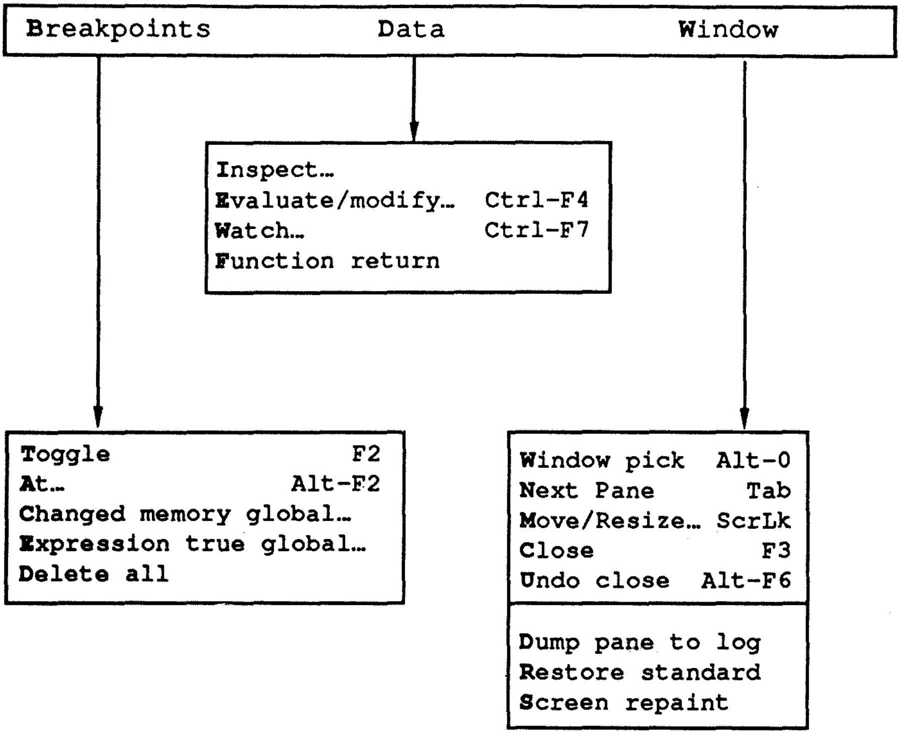
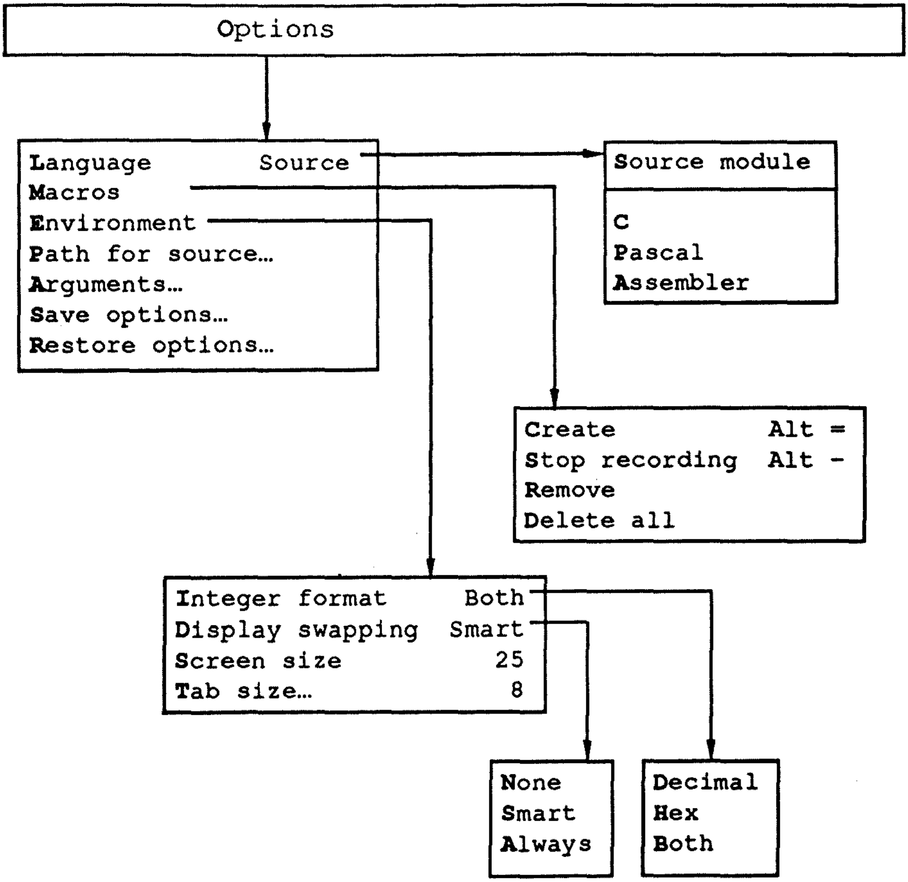


図12.3　Optionsメニュー



第13章

デバッグの進め方

デバッグは，プログラムの設計やコーディングの別の側面であると言うことができます。デバッグは，なかば科学，なかば技術です。問題の箇所を突き止めるために用いることができる特別な方法もありますが，ほんの少し勘を働かせるだけで，長時間かかる作業を大幅に短縮できることもあります。

デバッグの経験を積めば積むほど，プログラムの中の問題箇所をより短時間に突き止めることができるようになります。また自分に合ったデバッグのやり方を習得するとともに，しばしば問題を引き起こすプログラムの書き方を直していくことができます。

この章では，まず，正しく動作しないプログラムがあるとき，それをデバッグするにはどこから始めればよいかを説明します。次にいくつかの異なるデバッグの進め方について説明します。さらにプログラムの中で見つかる可能性のある，いろいろな種類のバグについて説明します。最後にプログラムが正しく動作することを確認するためのいくつかのテスト方法を示します。

プログラムが正しく動作しないとき

心配しないでください。最も熟練したプログラマでも，最初から正しく動作するプログラムを書くことは困難だからです。

多くの時間を費やしたあげくにそれが徒労に終わってしまうのを避けるために，バグがありそうなところを当てずっぽうに調べて見たくなる誘惑に負けないでください。もっとよいやり方は，「分割しながら絞り込む(いわゆる切りわけ)」という確実な手法を使うことです。

この手法は，仮定をしては試して見るという手順を繰り返しながら，問題箇所を絞りこんで行きます。たとえば「この問題は，関数***abc***の呼び出しの前に発生しているに違いない。」という仮定をした場合には，関数***abc***の呼び出しの前でプログラムの実行を停止して，その仮定が正しいかどうか調べます。その点で問題が見つかったときは，「その問題はさらに前のある箇所より前で発生する」という新たな仮定をすることができます。

反対に関数***abc***の呼び出しの前で問題が見つからなかったときは，最初の仮定が間違っていたことになります。この場合には，元の仮定を「この問題は，関数***abc***が呼び出された後のどこかで発生しているに違いない。」というように変更しなければなりません。このようなテストを繰り返すことにより，プログラムの問題を引き起こしている部分を，短時間で見つけることができます。

それではプログラムを停止させたときに，プログラムが正しく動作しているかどうかをどうやって調べたらよいでしょうか。プログラムの動作を調べる最もよい方法は，プログラム変数やデータ項の値を調べることです。たとえば配列をクリアするルーチンがあるとき，ルーチンが実行された後にプログラムを停止させ，配列の各要素がクリアされているかどうか確かめることにより，そのルーチンの動作を調べることができます。

デバッグスタイル

プログラムを書くスタイルが一人一人異なるように，誰もが自分自身のデバッグスタイルを発展させていきます。ここで説明するデバッグのやり方は，あなたが自分のスタイルを築きあげるための出発点にすぎません。

また多くの場合に，デバッグするプログラムの用途が，そのプログラムのデバッグのやり方に影響を及ぼします。プログラムの中には，自分で使うものや，特定の仕事を行なうために1回か2回しか使われないものもあります。このようなプログラムに対しては，プログラムのすべての構成要素からテストする全面的なテストは，おそらく時間の浪費でしょう。プログラムの出力を調べることにより，そのプログラムが正しく動作しているかどうかわかるときは特にそうです。他の人も使用するプログラムや，出力を調べるだけでは正しく動作しているかどうかの判断が困難なプログラムは，もっと厳密にテストをしたいでしょう。

プログラムの全体的な動作を調べる

簡単な使い捨てのプログラムに対する最もよいやり方は，プログラムを実行して〝何が起きるか見る〟ことです。もし問題が見つかったら，プログラムを再ロードして最も簡単な入力を与えて実行し，出力をチェックします。出力が正しければ，それよりも複雑な入力を与えて実行し，出力をチェックします。こうして，出力に誤りが発生するまでだんだん複雑な入力でテストします。このテストにより，プログラムが正しく動作する上限または下限を知ることができます。

プログラムの各部分からデバッグする

プログラムに絶対にバグが無いようにしたいときは，プログラムがいくつかのテスト入力データに対して期待されるとおりに動作することをチェックするだけでなく，個々のルーチンもテストする必要があります。それにはいくつかの方法があります。ひとつの方法は，各ルーチンを書いたときに，そのルーチンをテストするやり方です。ひとつのルーチンを書いたら，そのルーチンを呼び出してテストデータを与えるテストプログラムに，そのルーチンを組み込んでテストします。もうひとつの方法は，プログラム全体が完成した

あとに，デバッガを使って各ルーチンを実行してみることです。

バグの種類

プログラムのバグは，大まかに2つの種類にわけることができます。使用しているプログラミング言語(C, Pascal, アセンブラ)に特有なバグと，すべてのプログラミング言語に共通なバグです。プログラムをデバッグしているときに，いろいろなプログラミングの誤りを学ぶでしょう。この知識は，将来同じ誤りをおかすのを避けるために役に立ちます。また自分で書いたプログラムのバグを探すときに，どういうところにバグが存在する可能性が高いか，見当をつける助けになります。

一番大切なことは，バグを見つけたときに，それがどういう種類のバグであるかを理解するように努め，間違いのないコードを書く能力を向上させることです。バグを発見することが上手であるよりも，バグのないコードを書ける方がよいことです。

Cに特有のバグ

Turbo Cのユーザーズガイドには，C言語によるプログラミングにおいて陥りやすい落とし穴が説明されています。それらの落し穴についての説明を繰り返しさらに詳しく説明するのに，デバッグのやり方について説明するこの章よりも適当なところはないでしょう。

Turbo Cコンパイラは，他のコンパイラが警告を出さないような，多数のCに特有のバグを発見する能力が非常に優れています。コンパイラが発することのできるすべての警告をオンにすることにより，デバッグに要する時間を大幅に短縮することができます（これらの警告をオンにする方法については，Turbo Cのユーザーズガイドを参照してください)。

以下では，C言語によるプログラミングにおいて間違えやすい点を説明します。ただしこれ以外にも，いろいろと間違えやすい点があります。Turbo Cコンパイラは，これらの誤りのいくつかに対して警告メッセージを出します。警告メッセージが出されたときは，必らずその原因を調べるようにしてください。なぜなら，それらのメッセージが警告している箇所から，バグが生まれるかもしれないからです。

初期化されていない自動変数を使う

Cでは，関数の内部で宣言される自動変数は，なんらかの値を設定するまでは，どんな値になっているかわかりません。たとえば次の関数では，**while**ループが何回実行されるかを前もって知ることはできません。

```
do_ten_times ()
{
    int n;
    while (n < 10)
    {
        ...
        n++;
    }
}
```

なぜなら，*n*がカウンタとして使用される前にゼロに初期化されていないからです。

代入(=)と比較(==)を混同する

Cでは，等号(=)は代入演算子，二重等号(==)は左右のオペランドが等しいかどうかを調べる関係演算子です。代入演算子と関係演算子を間違えて次のように書くと，

```
if (x = y) {
    ...
}
```

変数*y*の値が変数*x*に代入され，*y*の値がゼロでないときは中カッコ内の文が実行されます。*x*と*y*が等しいかどうかを調べるには，(*X*=*y*)ではなく(*x*==*y*)と書かなければなりません。

演算子の優先順位を間違える

Cには非常に多数の演算子があるので，たくさんの演算子を含む式では，どの演算子が先に適用されるのかを間違えることがあります。たとえば次の式は12になります。

```
x = 3 << 1 + 1 ;
```

優先順位を間違えて左シフト演算子<<が先に適用されると考えると，*x*は7になるものと思い込んでしまいます。

ポインタの演算の間違い

ポインタを使って配列の要素を指し示すときは，ポインタに対する加算，減算に注意してください。たとえば次は，

```
int *intp;
intp = intp + sizeof(int);
```

期待していること（整数配列の次の要素を指すようにポインタ*intp*をインクリメントする）をしてくれません。*intp*は配列の要素2つ分進められます。なぜならCコンパイラは，ポインタに数を加えたり引いたりするとき，そのポインタが指すオブジェクトのサイズ(バイト数)を単位としてアドレスを増減するからです。ですから，ポインタ*intp*が配列の次の要素を指すようにしたいときは，次のように*intp*に1を加えるだけで済むわけです。

```
intp++ ;
```

符号が拡張されることを忘れる

整数をサイズの異なる整数に代入するときには，注意する必要があります。たとえば整数スカラーをそれよりもサイズの大きい整数にコピーしたときは，サイズの小さい整数の符号ビットが，サイズの大きい整数の高位の部分を通して最上位ビットまで埋められ，保存されます。次の例では，

```
int i = 0XFFFE;
long l;
l = i;

if (l & 0X80000000)
{
    ...                    /* 実行される */
}
```

int型整数*i*の値がlong型整数*l*に代入されるときには，-2(0xfffe)が-2(0xfffffffe)になります。したがって*l* & 0x8000000はゼロにならず，中カッコの中の文が実行されます。

Cでは，スカラー値(**char**，**int**など)同士であれば自由に代入ができます。このCの強力な機能は，結果がどうなるかをよく理解していないと，問題を引き起こすことがあります。

桁の切り詰めを忘れる

この間違いは，ひとつ前の間違いと原因が反対です。たとえば

```
int i;
long l = 0x10000000;
i = l;
while (i > 0) {
    ...                          /* 実行されない */
}
```

この例では，*l*の値が*i*にコピーされると，*l*の上位16ビットは破棄され，*i*の値はゼロ(0x0000)になります。したがって，**while**ループの文は一度も実行されません。

余分なセミコロン

次の**for**文は，一見正しいように見えます。

```
for (x = 0; x < 10; x++) ;
{
    ...                     /* 1回だけ実行される */
}
```

しかし，中カッコの中の文は1回しか実行されません。なぜでしょうか。よく調べると，**for**式の後のセミコロン(；)が問題であることがわかります。このバグのために，**for**ループは10回実行されますが，その間その下のブロックは実行されません。そして**for**ループの実行が終了すると，その下のブロックが一回実行されます。このバグはなかなかやっかいです。なぜなら，コードブロックの書式やインデントを調べる通常の手法では，発見が難しいからです。

副作用のあるマクロ

次の問題は，発見するのが非常に困難です。人によっては〝二度と＃defineマクロを使わない〟ことを誓うかもしれないほどです。たとえば，次のマクロでは，

```
#define  toupper(c)  'a'<= (c) && (c) <= 'z' ? (c)-'a'+'A' : (c)
char c, *p ;
c = toupper(*p++) ;
```

*p*は，*p*が指す文字により，2回インクリメントされる場合と3回インクリメントされる場合があります。このタイプの間違いは，副作用がマクロの定義の内部に隠されているため，見つけるのが非常に困難です。

自動変数名の重複

この間違いも発見が困難です。たとえば次のコードでは，

```
myfunc()
{
    int n;
    for (n = 5; n >= 0; n--)
    {
        int n = 10;
        ...
        if (n == 0)
        {
            ...                    /* けして実行されない */
        }
    }
}
```

自動変数*n*が外側のブロックと内側のブロックの両方で使われています。そのため，内側のブロックの*n*が外側のブロックのnへのアクセスを隠蔽しています。この間違いは，考える以上におかしやすいものです。なぜなら，ほとんどのプログラマは，ローカルなループカウンタ変数に，ごく少数の変数名（*i*や*n*など）しか使わないからです。

自動変数の使い方が間違っている

```
int *divide_by_3 (int n)
{
    int i;
    i = n / 3;
    return(&i);
}
```

この関数は，結果を指し示すポインタを返そうとしているように見えます。問題は，関数が値を返すまでに変数の値が無効になり，他のスタックデータによって上書きされてしまう可能性が高いことです。

関数が値を返さない場合がある

値を返す関数において，ある特別の場合に値を返すのを忘れることがあります。たとえば次の関数は，

```
char *first_capital_letter(char *p)
{
    while (*p)
    {
        if ('A' <= *p && *p <= 'Z')
            return(p) ;
        p++ ;
    }
}                                   /* 何も返さない */
```

文字列の中に大文字がない場合には，戻り値は無意味な値になります。ここでは，**while**ループの後に**return**(0)を入れる必要があります。

break文の使い方の間違い

break文では，**for**，**while**，**do..while**，**switch**文の中から1レベルしか抜け出せません。たとえば次のコードで，

```
for (...)
{
    while (...)
    {
        if (...)
            break;          /* ここでforループから抜け出したい */
    }
}
```

break文によって抜け出すことができるのは**while**ループだけです。ネストしたループ全体から抜け出すには，**goto**または**return**を使います。

何もしないコード

タイプミスの結果，まったく問題なくコンパイル可能であるが，意図していることをしてくれないソースコードになることがあります。たとえばa+=bと入力するつもりで

```
a + b ;
```

と入力してしまうことがあります。このコードは，実行されても何の変化も引き起こしません。タイプミスにも注意してください。

一般的なバグ

次に一般的なバグの例を示します。実際のデバッグにおいて出会う可能性のあるバグは多種多様であり，ここに説明したバグはほんの一部にすぎません。

隠れた副作用を忘れる

関数の呼び出しが副作用を引き起こすことがあります。たとえば次の例では，

```
char workbuf[20];
strcpy(workbuf,"all done¥n");
convert("xyz");
printf(workbuf);
...
convert(char *p) {
    strcpy(workbuf, p);
    while (*p)
    ...
}
```

配列*workbuf*に読み込まれた文字列が，**strcpy**関数によって壊されます。**strcpy**は専用のワークバッファを使うようにすべきです。

別のところで初期化されていると思い込む

別のルーチンにより，必要な初期化が行なわれていると思い込むことがあります。たとえば次の例は，

```
char *workbuf;
addworkstring(char *s)
{
    strcpy(workbuf, s);
}
```

*workbuf*にメモリが割り当てられていることを前提にしています。しかしまだ割り当てられていない場合には，*workbuf*がたまたま指している〝見当のつかない〟アドレスに，文字列が格納されます。これはエラーにはなりませんが(警告メッセージが出されます)，問題を引き起こす原因になり得ます。このルーチンに次のような文を付け加えると，メモリが割り当てられていない場合にも，そのような心配がなくなります。

```
if (workbuf == 0)  workbuf = (char *) malloc(20) ;
```

不要になったときにメモリ割り当てを解放しない

このタイプのバグのために，ヒープ領域が足りなくなってプログラムがクラッシュするまでには，長い時間がかかることがあります。たとえば次の例では，

```
crunch_string(char *p)
{
    char *work = (char *) malloc(strlen(p));
    strcpy(work, p);
    ...
    return (p);
}
```

return文の前で，ポインタ*work*が指す文字配列に割り当てた領域を解放しておかなければなりません。

フェンスポスト(柵の支柱)エラー

フェンスポストエラーという名前は，「100mの柵を作るとき，10mごとに支柱を立てると，支柱は何本必要か」という昔からの問題に由来しています。何本必要でしょうか。早とちりして10本と答えないでください(向こうの端の最後の支柱を忘れています)。プログラムを書くときも，同じような間違いをすることがあります。Cによるプログラミングの例を示します。

```
for (n = 1; n < 10; n++)
{
    ...
}
```

この例では，数字の1と10がすぐに目に入ります。そこで**for**ループが10回実行されるものと，勘違いすることがあります。正しく動作するようにするには，関係演算子<を<=に変えるのがいいでしょう。

Pascalに特有のバグ

Pascalの強力な型チェックおよびエラーチェック機能のおかげで，Pascalに特有のバグは僅かです。ただしTurbo Pascalでは，そのチェック機能の多くをオフにすることができるようになっているので，チェック機能をオフにしたときは，オンにしておけば避けられたはずの間違いが入りこむ可能性があります。次にPascalにおいて陥りやすい落とし穴について説明します。

初期化されていない変数を使う

Turbo Pascalでは，変数は初期化されません。代入文により値を設定するか，型つき定数として宣言するかのどちらかにより，自分で初期化しなければなりません。次の例を見てください。

```
program Test;
var
  I, J, Count : integer;
begin
  for I := 1 to Count do begin
    J := I*I;
    Writeln(I:2, ' ', J:4)
  end
end.
```

この例の変数*Count*は，生成されたとき，たまたまメモリのその場所に入っていた値を持つことになります。そのため，このループが何回実行されるかはまったくわかりません。

また手続きや関数の内部で宣言された変数は，そのルーチンに入るたびに生成され，そのルーチンから出るときに，それらの変数に割り当てられていたメモリが解放されます。ですから，それらの変数がその前の値を保持していることを当てにすることはできません。

初期化していないポインタを使用する

ポインタに関してよくやる間違いが3つあります。1つ目は，先に説明したように，ポインタにアドレス（**nil**またはその他の値）を設定する前に，そのポインタを使うことです。他のすべての変数やデータ構造と同じように，ポインタも宣言しただけでは初期化されません。*New*手続きに渡すか，**nil**を代入することにより，できるだけ早く初期値をセットすべきです。

2つめは，**nil**ポインタを参照すること，すなわちポインタそれ自体の値が**nil**のときに，そのポインタが指すデータにアクセスしてしまうことです。たとえば，ポインタにより連鎖状に連結された複数のレコードがあり，ある値を持つレコードを見つけるためにそのリストを探したいものとします。そのための関数は，次のようになるでしょう。

```
function  FindNode(Head : NodePtr; Val : integer) : NodePtr;
var
  Temp : NodePtr;
begin
  Temp := Head;
  while (Temp^.Key <> Val) and (Temp <> nil) do
    Temp := Temp^.Next;
  FindNode := Temp;
end;  { of function FindNode }
```

このコードでは，*Val*がこのリストのどのノードの*Key*フィールドの値にも等しくない場合に，ポインタ*Temp*の値が**nil**になり，次に*Temp^.Key*の値が評価されます。その結果がどうなるかはわかりません。これを解決するにはどうすればよいでしょうか。**while**文を次のように書き換えます。

```
while (Temp <> nil) and (Temp^.Key <> Val) do
```

さらにTurbo Pascalの{**$B-**}オプションまたは**Options/Compiler/Boolean evaluation**コマンドを使って，論理式のショートサーキット評価をオンにします。このようにすると，*Temp*が**nil**になったときは，2番目の手続きは評価されません。

3つめは，ポインタを***Dispose***または***FreeMem***手続きに渡したら，ポインタが**nil**にセットされると思い込むことです。動的変数が破棄された後も，ポインタの値は元のままです。しかしポインタが指し示しているメモリ領域はヒープに返されます。ポインタの指し示すデータ構造を破棄したら，ポインタの値はnilにセットすべきです。

スコープを間違える

Pascalでは，手続きや関数を非常に深くネストすることが許されています。そしてネストした手続きや関数の内部で，それぞれ，ローカルな変数を宣言することができます。次の例を見てください。

```
program Confused;
var
  A, B, T : integer;

  procedure Swap(var A,B : integer);
  var
    T : integer;
  begin
    Writeln('2: A,B,T = ', A:3, B:3, ' ', T);
    T := A;
    A := B;
    B := T;
    Writeln('3: A,B,T = ', A:3, B:3, ' ', T)
  end;  { of procedure Swap }

begin { main body of Confused }
  A := 10; B := 20; T := 30;
  Writeln('1: A,B,T = ', A:3, B:3, ' ', T);
  Swap(B,A);
  Writeln('4: A,B,T = ', A:3, B:3, ' ', T)
end.  { of program Confused }
```

このプログラムの出力は，どうなるでしょうか。次のようになるはずです。

```
1: A,B,T =  10 20 30
2: A,B,T =  20 10 22161
3: A,B,T =  10 20 20
4: A,B,T =  20 10 30
```

この例で示しているのは，変数*A*, *B*, *T*が2組生成されるということです。プログラムのメインボディの中で使用されているのは，グローバルな変数です。それに対して*Swap*の仮引数*A*, *B*と*Swap*の内部で宣言されている変数Tは，*Swap*にローカルな変数です。さらに混乱を招くことには，*Swap*を呼び出すときに*Swap*(*B*, *A*)としています。これは，仮引数*A*に渡されるのはグローバル変数*B*であり，仮引数*B*に渡されるのはグローバル変数*A*であることを意味します。グローバル変数の*T*とローカル変数のTには，なんの関係もありません。

この例には本当のバグはありません。しかし，値の変更されない変数が変更されるものと勘違いすると，問題が起きる可能性があります。たとえば，メインボディの中の変数*T*の値は変化していませんが，変化すると考えた人がいるかもしれません。このバグは，先に説明した〝隠れた副作用〟を引き起こすバグと反対です。

次のようなレコード型を宣言した場合には，さらにまぎらわしくなります。

```
type
  RecType = record
              A,B : integer;
            end;

var
  A,B : integer;
  Rec : RecType;
```

with文の内部では，*A*または*B*の参照は，変数ではなく，レコード***RecType***のフィールドの参照と解釈されます。

余分なセミコロン

Cと同じようにPascalでも，空文(1個のセミコロンだけからなる文)が許されています。セミコロンは，間違った場所に置かれると，あらゆる種類の問題を引き起こします。次のプログラムを見てください。

```
Program Test;
var
  I,J : integer;
begin
  for I := 1 to 20 do ;
  begin
    J := I * I;
    Writeln(I:2,' ',J:4)
  end;
  Writeln('All done!')
end.
```

このプログラムの出力は，1から20までの整数とそれらの2乗のリストではなく，ただ次のようになります。

```
20 400
All done!
```

この原因は，**for**文の**do**の後ろにセミコロンがついていることです。そのため，空文が20回実行されます。その後で**begin..end**ブロックの中の文が実行され，さらに最後の***Writeln***文が実行されます。これを修正するには，**do**キーワードの後のセミコロンを除去します。

関数名に値が代入されない場合がある

関数から出るときは，必らず関数名に値を代入しなければなりません。次の例を見てください。

```
const
  NLMax = 100;
type
  NumList = array [1..NLMax] of integer;
  ...
function FindMax(List : NumList; Count : integer) : integer;
var
  I, Max : integer;
begin
  Max := List[1];
  for I := 2 to Count do
    if List[I] > Max then
    begin
      Max := List[I];
      FindMax := Max
    end
end;  { of function FindMax }
```

この関数は，配列*List*の中の最大値が*List*［*1*］にない限りは問題なく動作します。最大値が*List*［*1*］にあるときは，関数名*FindMax*に値が代入されません。この関数を修正すると，次のようになるでしょう。

```
begin
  Max := List[1];
  for I := 2 to Count do
    if List[I] > Max then
      Max := List[I];
    FindMax := Max
end;  { of function FindMax }
```

word型またはbyte型の変数をデクリメントする

符号なしスカラー(byte, word)をデクリメントしながら，関係演算子(>=)でテストしないように注意してください。たとえば次のコードは，無限ループを引き起こします。

```
var
  w : word;
begin
  w := 5;
  while w >= 0 do
    w := w - 1;
end.
```

この例では，**while**文の5回目の繰り返しの後で*w*が0になります。その次に*w*から1が引かれると，*w*は65535になります（word型の値の範囲は0から65535だからです）。そして，やはり*w*>=*0*の条件は真になります。このような場合には，integer型かlongint型を使うべきです。

境界や特殊な場合を忘れる

前の例の関数*FindMax*は，修正する前のものも，修正したものも，*Count*>=1であると仮定していることに注意してください。しかし，*Count*=0（リストが空）になる可能性もあります。もしこの状態で関数*FindMax*を呼び出すと，たまたま*List* [*1*] に入っていたものを返します。また*Count*>*FindMax*となる可能性もあります。この場合には，実行時エラーになるか（範囲チェックコードの生成をオンにしてコンパイルされているとき），*List*に含まれていないメモリ位置まで探すことになります。

この問題に対しては2つの解決策があります。1つは，*Count*が*1..NLMax*の範囲外のときは，*FindMax*を呼び出さないことです。ふざけて言っているのではありません。ルーチンを呼び出すための条件をきちんと決めることは，ソフトウェアの設計の良否を決める重要な要素の1つです。

もう1つの解決策は，*Count*の値を調べて，*1..NLMax*の範囲外のときには何か特定の値を返すようにすることです。たとえば，*FindMax*のメインボディを次のように書き直します。

```
begin
  if (Count < 1) or (Count > NLMax) then
    Max := -32768
  else
  begin
    Max := List[1];
    for I := 2 to Count do
      if List[I] > Max then
        Max := List[I]
  end;
  FindMax := Max
end;  { of function FindMax }
```

ただしこのようにすると，次に説明するPascalの代表的な落とし穴（範囲の誤り）に陥る可能性があります。

範囲エラー

Turbo Pascalの範囲チェックコードの生成を制御するスイッチ指令{$R}は，デフォルトでオフになっています。そのため，範囲チェックコードを生成する場合に比べて高速でコンパクトなコードを生成することができます。しかしそのかわりに，変数に許容範囲外の値を代入したり，前例に示すように配列の存在しない要素を参照したりする誤りの余地が生まれます。

この種の誤りを見つけ出すための第1のステップは，プログラムの中に{$**R**+}コンパイラオプションを挿入し，それをコンパイルして再実行することです。誤りがどこにあるかわかっている(疑っている)ときには，その部分の前にこのスイッチ指令を置き，その部分の後に{$**R**-}指令を置いて，その部分に対してだけ範囲チェックをオンにすることができます。範囲エラーが起きたときはプログラムは実行を停止し，エラーメッセージを表示して，エラーの発生したアドレスを示します。

よく起きる範囲エラーの1つは，**while**ループまたは**repeat**ループを用いて，配列の要素を順番に調べるときに発生します。たとえばある値を含む配列の要素を探しているとしましょう。そしてその値が見つかったときはその要素のインデックス（添字）を返し，配列の最後に到達したときはゼロを返したいとします。このようなとき最初に書くプログラムは，次のようなものでしょう。

```
function FindVal(List : NumList; Count,Val : integer) : integer;
var
  I : integer;
begin
  FindVal := 0;
  I := 1;
  while (I <= Count) and (List[I] <> Val) do
    Inc(I);
  if I <= Count then
    FindVal := I
end;  { of function FindVal }
```

なかなかよくできているように見えます。しかし，論理式の完全評価を指定してコンパイルした場合には，*Val*が*List*の中に見つからないときには，実行時エラーになってしまいます。なぜでしょうか。**while**ループの条件式が最後にテストされるとき，*I*は*Count*+1になります。したがって*Count=NLMax*の場合には，*List*の添字の範囲を越えることになるからです。

アセンブリ言語に特有のバグ

アセンブリ言語のプログラミングでも，陥りやすい落とし穴がいろいろあります。次にそれらの落とし穴のいくつかについて説明します。これらの誤りに関するより詳しい説明と，それを避けるための助言については，Turbo Assemblerのユーザーズガイドを参照してください。

DOSに戻るのを忘れる

CやPascal，その他の言語では，プログラムの中に明示的な終了コマンドが書かれていなくても，実行するコードがなくなるとプログラムは自動的に終了してDOSに戻ります。しかしアセンブリ言語ではそうはいきません。アセンブリ言語では，明示的に要求したことしかやってくれないからです。DOSに戻るコマンドのないプログラムを実行すると，プログラムのコードの末尾に達しても実行が停止せず，その後のメモリにたまたま存在するコードが実行されます。

RET命令を忘れる

サブルーチンの実行が終了した後は，必らず呼び出し側のコードに戻らなければなりません。各サブルーチンに**RET**命令を入れるのを忘れないでください。プログラムを入力するときに**RET**命令を忘れやすいので注意してください。

生成されるリターンのタイプが間違っている

PROC疑似命令には2つの働きがあります。ひとつは，プロシージャの名前を定義することです。もうひとつは，そのプロシージャをnearプロシージャまたはfarプロシージャのどちらにするか制御することです。

ではプロシージャの中の**RET**命令のタイプは，プロシージャのタイプ(nearまたはfar)と同じでいいでしょうか。

答えはイエスでもあり，ノーでもあります。問題が起きるのは，複数のサブルーチンを同一のプロシージャにまとめた場合です。なぜなら，これらのサブルーチンにはそれ自体の**PROC**疑似命令がなく，各サブルーチンの**RET**命令のタイプはそのプロシージャのタイプと同じになりますが，そのタイプが個々のサブルーチンに対しては正しくない場合があるからです。

オペランドの順番が逆になっている

多くの人々とって，8086アセンブリ言語の命令のオペランドの順番は，実際とは逆であるように見えるようです（この見方にも確かに理由はあります)。次の命令が，

```
mov   ax,bx
```

〝move AX to BX（AXからBXへ動かす）〟を意味するのであれば，この行は書かれているとおりに左から右に見ていくことができます。そして多くのマイクロプロセッサメーカーが，自社のアセンブリ言語の設計にこの順番を採用しています。しかしインテル社は，8086アセンブリ言語にそれとは異なる順番を採用しました。そのため上の命令は〝move BX to AX (BXからAXへ動かす)〟を意味します。そしてこのことが，しばしば混乱を引き起こします。

スタックスペースを確保するのを忘れる，または確保したスペースが小さすぎる

たいていの場合，スタックのために明示的にスペースを割り当てないのは，薄氷の上を歩くようなものです。スタックスペースが確保されていないプログラムは，あるときは動作したとしても，あらゆる条件のもとで正しく動作するという保証はありません。ほとんどのプログラムでは，**.STACK**疑似命令により，スタックのためのスペースを確保しておかなければなりません。プログラムが使用する可能性のある最も深いスタックに対して，十分である以上のスペースを確保しておく必要があります。

必要なレジスタを破壊するサブルーチンを呼び出す

アセンブラコードを書いているときは，レジスタを，そのとき書いているプロシージャだけが使用できるローカルな変数であると思い込みやすいものです。特に他のプロシージャを呼び出してもレジスタは変化しないと思い込む傾向があります。そうではありません。レジスタはグローバル変数です。そして各プロシージャは，一部あるいはすべてのレジスタを，保存することも破壊することもできます。したがってプロシージャを呼び出すときは，レジスタが破壊される可能性があることに注意する必要があります。

条件処理でジャンプ命令またはジャンプ条件を間違える

アセンブリ言語には多数の条件ジャンプ命令（**JE, JNE, JC, JNC, JA, JB, JG**, etc.)が用意されており，コードを書く上で非常に柔軟性がありますが，そのかわり間違ったジャンプ命令を選択する可能性も高くなります。

またアセンブリ言語による条件処理には，少なくとも2行必要です。1行は比較のため，もう1行は条件ジャンプ命令のためです。複雑な条件に対しては，もっと多数の行が必要に

なります。そのためアセンブリ言語による条件処理は，CやPascalによる条件処理に比べて，直感的にわかりにくく間違えやすいので，特に注意する必要があります。

REPつきのストリング命令で文字列をオーバーランする

ストリング命令の働きには，注意すべき点があります。それは，命令が実行された後に，その命令が使用するポインタが，直前にアクセスしたアドレスから1バイト(ワード命令の場合には2バイト)進められることです。このことは，繰り返しプレフィクス付きのストリング命令，特に**REP SCAS**や**REP CMPS**において，範囲エラーをおかす原因になることがあります。

CXがゼロのとき何もしないことを忘れて
セグメント全体がカバーされると思い込む

繰り返しプレフィクス付きのストリング命令は，CXがゼロのときは何もしません。この機能により，繰り返しプレフィクス付きのストリング命令を実行する前に，CXがゼロかどうかチェックする必要がなくなります。そのかわりバイトサイズのストリング命令では，セグメントの全バイトにアクセスすることができません。

ディレクションフラグの設定が間違っている

ストリング命令が実行されたときは，ディレクションフラグの状態により，ポインタSIまたはDIまたはその両方が，インクリメントまたはデクリメントされます。

ディレクションフラグは，**CLD**によってクリアすることができます。また**STD**によってセットすることができます。ディレクションフラグがクリアされているときは，ストリング命令のポインタがインクリメント（カウントアップ）され，セットされているときはデクリメント(カウントダウン)されます。ディレクションフラグは，いったんクリアまたはセットされると，別の**CLD**または**STD**が実行されるまで，あるいは**POPF**または**IRET**によってスタックからポップされるまで，同じ状態のままです。ディレクションフラグを一度設定すると，同じ方向に動作するストリング命令をいくつでも実行できることは大変便利です。しかしこれは，ずっと先に実行されたコードのためにストリング命令が反対の方向に動作する発見が困難なバグの原因にもなります。

文字列比較に間違った比較命令または間違ったプレフィクスを使用する

CMPS命令は，メモリの2つの領域を比較します。それに対してSCAS命令は，アキュムレータをメモリ領域と比較します。REPEプレフィクスを付けたときは，どちらの命令も，CXレジスタがゼロになるか，不一致が検出されるまで，比較を繰り返します。

REPNEプレフィクスを付けたときは，どちらの命令も，CXレジスタがゼロになるか，一致が検出されるまで，比較を繰り返します。CMPS命令とSCAS命令，REPEプレフィクスとREPNEプレフィクスの意味を，よく取り違えることがあります。

ストリング命令のデフォルトセグメントを忘れる

各ストリング命令は，ソースオペランド(もしあれば)はDSセグメントに，デスティネーションオペランド(もしあれば)はESセグメントにあるものとみなします。このことを忘れて，たとえばSTOSBをデータセグメントに対して実行しようとすることがあります。なぜなら，ストリング命令以外の命令で操作しているすべてのデータは，通常データセグメントにあるからです。

BYTE操作からWORD操作への変換に誤りがある

一般的に，ストリング命令に対しては，可能な限り大きなデータサイズ（通常はWORD，80386ではDWORD）を使う方が有利です。なぜならストリング命令による処理は，データサイズが大きいほど速いことが多いからです。

しかしいくつかの落とし穴もあります。ひとつは，単純に次の命令

```
shr  cx,1
```

によってバイト数をワード数に変換しようとすると，CXが奇数のときは1バイト少なくなることです。なぜなら，最下位のビットがシフトアウトされるからです。

もうひとつは，ワードサイズのストリング命令，たとえばSTOSWに対して，ワード数ではなくバイト数を，繰り返し回数としてセットすることです。この間違いは，他のデータを破壊し，あらゆる問題を引き起こします。

複数のプレフィクスを使用する

ストリング命令に複数のプレフィクスを付けると，誤りが発生する可能性が高くなるので，通常は避けてください。

ストリング命令に対するオペランドの指定が有効と思い込む

ストリング命令のオプションのオペランドは，データサイズの指定とセグメントオーバーライドのために使用されます。他のオペランドを指定したとき，オペランドによって参照指示されるメモリ位置が実際にアクセスされるかどうかは保証されません。

乗算によりレジスタを破壊する

乗算は，8ビット×8ビット，16ビット×16ビット，または32ビット×32ビットのどれでも，ソースオペランドとして使用されるアキュムレータ（またはその1部）以外の少なくとも1つのレジスタの内容を破壊します。

ストリング命令がいくつかのレジスタの値を変えるのを忘れる

ストリング命令**MOVS**，**STOS**，**LODS**，**CMPS**，**SCAS**は，いくつかのフラグと3つのレジスタに影響を及ぼす可能性があります。ストリング命令を使うときは，SIまたはDIまたはその両方が，ディレクションフラグの状態により，インクリメントまたはデクリメントされることを忘れないでください。また**REP**プレフィクスを付けたストリング命令が実行されると，CXが，ゼロになるまで少なくとも1回デクリメントされることに注意してください。

フラグを変化させない命令であるのに変化させると勘違いする

レジスタやフラグに影響を及ぼさないように見えるのに，実際には影響を及ぼす命令があります。また，影響を及ぼしそうに見えるのに，どのフラグにも影響を及ぼさない命令もあります。

フラグを利用する命令がフラグがセットされる場所から離れすぎている

フラグが同じ状態を保持していることを保証できるのは，そのフラグに影響を及ぼす次の命令までです。そして通常，そのような命令はそんなに離れていないところにあります。ですから，フラグがセットされる(可能性のある)場所からできるだけ近いところで，そのフラグをテストして利用するのがよい習慣です。このようにすると，フラグを利用する命令が離れすぎている場合に発生する可能性のある，いろいろなバグを避けることができます。

メモリ参照オペランドと即値オペランドを間違える

アセンブリ言語では，命令のオペランドに変数のオフセットを指定したいときと，変数の値(その変数に格納されている値)を指定したいときがあります。変数のオフセットを指定するときは，変数名の前に**OFFSET**演算子を付けなければなりません。プログラムを書いているとき，変数のオフセットを指定するつもりでいて**OFFSET**演算子を忘れることがあるので注意が必要です。

セグメントの終わりから先頭に戻る

8086のプログラミングで難しいことのひとつは，メモリ全体を一連の長いバイトの配列としてアクセスすることができず，セグメントレジスタをベースとする64Kの範囲内でしか連続的にアクセスできないことです。セグメントは，発見が困難なバグをもたらします。なぜなら，プログラムがセグメントの終わりを越えて範囲外のアドレスにアクセスしようとしたときは，セグメントの終わりから先頭に戻って，再びそのセグメントの先頭にアクセスすることになるからです。

割り込みハンドラですべてのデータを退避していない

割り込みハンドラも，すべてのレジスタの内容を退避するようにするべきです。そのハンドラが変更するレジスタのみを明示的に退避することもできますが，割り込みハンドラに入ったときにすべてのレジスタをスタックにプッシュし，出るときにすべてのレジスタをポップするようにすることは〝有利な保険〟といえます。

オペランドおよびデータテーブルにグループオーバーライドを忘れる

GROUP疑似命令によってセグメントをグループ化すると，複数のセグメントの中の領域に，そのグループの先頭アドレスからのオフセットでアクセスすることができます。他のセグメントの領域にアクセスするために，セグメントレジスタにセグメントアドレスをロードする必要はありません。

困ったことに，マイクロソフト社のマクロアセンブラ(MASM)がセグメントグループを扱う方法には，多少問題があります。そのためTurbo Assemblerが出現するまでは，セグメントのグループ化は，アセンブリ言語では非常にやっかいなものでした。しかしセグメントのグループ化は，たとえやっかいでも，避けて通ることのできないものでした。なぜなら，アセンブラコードをCのような高級言語にリンクするために必要だからです。

MASM Quirksモードでは，Turbo AssemblerはMASMを完全にエミュレートします。このことは，MASM QuirksモードではTurbo AssemblerにもMASMがかかえているのと同じセグメントグループに関する問題があることを意味します。MASM Quirksモードを使うつもりのない人は，これ以上何も読む必要はありません。MASM Quirksモードを使おうと思っている人は，Turbo Assemblerのユーザーズガイドの詳しい説明を読んでください。

動作テスト

プログラムに有効な入力を与えて動作を調べることは，テスト作業の一部にすぎません。以下の節には，プログラムやルーチンに完全健康証明書を付与する前に，すべてのプログラムやルーチンに対して実施すべき重要なテストをいくつか説明します。

範囲の境界および限界をテストする

あるルーチンがある範囲の入力数で動作すると判断したら，その範囲の上限または下限の入力数でテストしてみるべきです。たとえば項目数が1から20までの長さのリストを表示するルーチンなら，項目が1個および20個のどちらの場合にも正しく動作することを確認します。このテストにより，ひとつ少なすぎたり，ひとつ多すぎたりする〝フェンスポスト〞エラーを発見することができます。

間違ったデータを入力してみる

あるルーチンが，有効な入力範囲全体に対して動作すると判断したら，無効な入力に対しても正しく動作することをチェックするべきです。無効なデータ，特に有効なデータに非常に近いデータが拒絶されるかどうかチェックする必要があります。たとえば1から20までの値を受入れるルーチンなら，0と21が拒絶されることを確認します。

何も入力しない

データが入力されない可能性は，プログラムを設計するときにも，プログラムをテストするときにもしばしば見過ごされます。プログラムを書くときに，データが入力されない場合のデフォルトの動作を決めておくようにすると，非常に使いやすいプログラムになります。

プログラム設計の一部としてのデバッグ

プログラムの設計を始めるときに，デバッグ段階の計画を立てることができます。プログラムの設計における最も基本的なトレードオフのひとつは，プログラムの各部分が，有効な入力を受け取っていること，および自分の出力が適正であることを，どの程度までチェックするようにするかということです。

たくさんのチェックを行なうようにすると，頻繁にエラー状態を知らせるが，何か適当な回復操作を行なうと動作を続ける非常に回復力のあるプログラムになります。そのかわり，プログラムのサイズが大きくなり，動作も遅くなります。このタイプのプログラムは，一般にデバッグがかなり容易です。無効なデータによるエラーが伝播する前に，各ルーチンが，データが無効であることを知らせるからです。

それとは逆に，入力や出力の有効性チェックをほとんど（あるいはまったく）行なわないプログラムを書くこともできます。このようなプログラムは，サイズが小さくなり，動作も高速になります。しかし各ルーチンへの入力データの間違いやちょっとしたバグのために，大いに悩まされる可能性があります。このタイプのプログラムは，最もデバッグが困難です。小さな問題が，実行中ずっと後の方では大きく拡大するからです。そのため，エラーを根源まで追跡することが困難です。

ほとんどプログラムは，この2つの手法を混用したものになります。

またユーザのキー入力や，ディスクファイルからの読み込みのような外部からの入力を，内部ルーチンから渡されるデータよりも，ずっと強い疑いをもって扱わなければならないこともよくあります。

サンプルデバッグセッション

このサンプルデバッグセッションでは，これまでの節で説明した手法を使ってデバッグを進めます。デバッグするプログラムは，第3章で使った例題プログラム（TCDEMO.C, TPDEMO.PAS）にわざとバグを入れたものです。

カレントディレクトリに，このデバッグセッションに必要な次の2つのファイルが入っていることを確認してください。Turbo Cのプログラムをデバッグするときは TCDEMOB.C と TCDEMOB.EXE，Turbo Pascalのプログラムをデバッグするときは TPDEMOB.PAS と TPDEMOB.EXE です（ファイル名の最後の*B*は〝buggy－虫がいる〟を表わします）。

Cのデバッグセッション

この節では，Turbo Cの例題プログラムを用いたデバッグセッションを説明します。

Turbo Pascalの例題プログラムを使ったデバッグセッションは，次の節に説明してあります。

間違いを探す

デバッグセッションを始める前に，バグのある例題プログラムを実行して，どこがおかしいか見てみましょう。次のように入力し**リターン**を押して，例題プログラムを起動してください。

```
TCDEMOB
```

すると例題プログラムは，テキストを1行ずつ入力するように求めます。次のように2行入力してください。

```
one two three
four five six
```

最後に何も入力しないで**リターン**押して入力を終了させてください。TCDEMOBは，入力したテキストについての分析結果を表示します。

```
Arguments:
Enter a line (empty line to end): one two three
Enter a line (empty line to end): four five six
Enter a line (empty line to end):
Total number of letters = 7
Total number of lines = 6
Total word count = 2
Average number of words per line = 0.3333333
'E' occurs 1 times, 0 times at start of a word
'F' occurs 1 times, 1 times at start of a word
'N' occurs 1 times, 0 times at start of a word
'O' occurs 2 times, 1 times at start of a word
'R' occurs 1 times, 0 times at start of a word
'U' occurs 1 times, 0 times at start of a word
There is 1 word 3 characters long
There is 1 word 4 characters long
```

Total number of letters(総文字数)，Total number of lines(総行数)，Total word count

(総語数)の数値が間違っていることに注目してください。その下の文字と単語の出現頻度表も間違っているように見えます。これは，プログラムに複数の間違いがあることを示唆する典型的な症状です。デバッグは，しばしばこのような状態から開始することになります。

どの問題から取りかかるかを決める

まずやるべきことは，どの問題から取りかかるかを決めることです。そのためのよい指針としては，〝最初〃に発生するように見える問題から取りかかることです。

このプログラムは入力行をまず単語に分解し，それから分析します。さらに空行が入力されると，入力の要求を停止して表を表示します。表だけでなく語数と文字数も間違っているので，単語に分解し文字数や語数を数える最初の処理段階に何か間違いがあると見なすことは，かなり確かな推測でしょう。

どの問題から取りかかるか大まかな計画が決まったので，いよいよデバッグを開始することができます。デバッグの進め方としては，まず*makeintowords*ルーチンが，入力行を正しく単語に分解するかどうかを調べます。次に*analyzewords*ルーチンが，各行を正しく分析するかどうかを調べます。

Turbo Debuggerを起動する

デバッグセッションを開始するには，次のように入力して**リターン**を押してください。

```
TD TCDEMOB
```

するとTurbo Debuggerが起動して，バグのある例題プログラムをロードし，初期画面を表示します。このデバッグセッションから抜け出してDOSに戻りたいときは，***GRPH-X***を押すといつでも抜けられます。どこかに迷い込んでどうすることもできなくなったときは，***CTRL-F2***を押すと，例題プログラムを再ロードして最初からやり直すことができます（この方法でデバッグセッションを再開したときは，ブレークポイントやWatchesウィンドウ内の設定は解除されないことに注意してください)。

最初にやることは*makeintowords*の動作を調べることなので，まずプログラムをこのルーチンの直前まで実行して，それから*makeintowords*の動作を調べます。それには2つのやり方があります。ひとつは*makeintowords*のソースコードを1行ずつ実行しながら調べる方法です。もうひとつは*makeintowords*の全体を一度に実行して，その結果から動作を調べる方

法です。

*makeintowords*のやるべきことは明確に決められているので，*makeintowords*が生成する出力バッファの内容を調べるだけで，正しく動作しているかどうか容易に判断できます。ですからここでは2番目のやり方を使いましょう。

まず例題プログラムの行41にカーソルを移動して**F4**を押し，行41まで実行してください。例題プログラムの画面が現われたら，次のように入力して**リターン**を押してください。

```
one two three
```

Inspectコマンドで調べる

プログラムが停止したのは，*makeintowords*が呼び出された後のソース行です。ここで*makeintowords*の動作が正しいかどうかを調べるために*buffer*の内容を見てみましょう。カーソルを1行上に移動して*buffer*に置いて**GRPH-F10 I**（Inspectコマンドを選択）を押してください。するとInspectorウィンドウが開いてbufferの内容が表示されます。矢印キーを使って，配列*buffer*の要素の表示をスクロールしてください。*makeintowords*は，各単語の最後にヌル文字(0)を1つ付加していることに注意してください。これは*makeintowords*が正しく動作していることを示します。したがってプログラムをさらにもう少し実行して，*analyzewords*の動作を調べる必要があります。

ESCを押してInpectorウィンドウを閉じてください。それから**F7**を2回押して*analyzewords*の先頭まで実行してください。

カーソルを*bufp*の下に移動して**GRPH-F10 I**を押し，*analyzewords*を呼び出すときの引数が*buffer*の内容を指し示しているかどうかを調べてください。そうすれば，*bufp*が確かにヌル文字で終了する文字列'one'を指していることがわかるでしょう。**ESC**を押して，Inspectorウィンドウを閉じてください。

どうも文字数と単語数の計算に問題があるように見えます。次に文字数と単語数を数える箇所に，ブレークポイントを設定しましょう。カーソルを行92に移動して**F2**を押し，ブレークポイントを設定してください。同じようにして，行96にもうひとつのブレークポイントを設定してください。さらにこの関数が返す文字数を見ることができるように，行98にブレークポイントを設定してください。このように複数のブレークポイントを設定することは，データが正しい順序で処理されているかどうかを調べたり，プログラムの中の重要なデータの値を調べるための代表的な方法です。

F9を押してプログラムを実行してください。プログラムは，行92のブレークポイントに到達して停止します。ここで*charcount*の値を見てみましょう。またこのブレークポイント

に到達するたびに*charcount*の値を見ることにしましょう。ですからここで，Watchコマンドを使って変数*charcount*をWatchesウィンドウの中に入れておきましょう。カーソルを*charcount*に移動して***CTRL-F10 W***を押してください。すると画面下部のWatchesウィンドウの中に，変数*charcount*とその現在の値0が表示されます。文字数が正しく数えられていることを確認するために，***F7***を押して1行実行してください。Watchesウィンドウには*charcount*が1であることが表示されます。

F9を押して，もう一度プログラムを実行してください。プログラムは，次の文字を読み込む行92に戻って停止します。***F9***をさらに2回押して，単語oneの最後の文字と終了文字であるヌル文字を読み込んでください。いま変数*charcount*の値は正しく3を指しているはずです。また配列*wordcounts*は，新しい単語を語数に加え入れるために，更新される直前です。ここまではすべて正常です。

ここでもう一度F9を押して，バッファ*buffer*内の次の単語の処理を開始させてください。何か変です。次の単語を処理したときにも，プログラムが行92で停止することを期待されたでしょう。でも停止しないで，この関数から戻るプログラム文まで進んでしまいました。実行が行98まで進んで停止するのは，行82から始まった**while**ループの条件式の値が，真(1)でなくなった場合だけです。これは＊*bufp* != 0が偽(0)になったことを示します。＊*bufp*!= 0が偽であるかどうかを調べてみましょう。行82に戻り，アスタリスク'＊'にカーソルを置いて***INS***を押し，次に')'の前の'0'にカーソルを置いて***INS***を押して，式＊*bufp*! =0の全体を指定します。そして***GRPH-D E***と入力して**Data**メニューの**Evaluate/modify**を選択し，**リターン**を押して指定して，この式を評価します。式の値は実際に0です。***ESC***を2回押して**Module**ウィンドウに戻ってください。

バグの発見と修正

バグを解決するための有力な手がかりが得られました。*bufp*が0を指し示している理由は，1つの単語の処理が終わると，行82から始まる内側の**while**ループから抜け出すからです。引き続いて次の単語を処理するには，その前の単語の終了文字である0を通り越すように*bufp*をインクリメントしなければなりません。それには行96の前に*bufp*＋＋という文を付け加える必要があります。ソースプログラムにこの文を付け加えて再コンパイルすることもできますが，Turbo Debuggerには，プログラムの中に式を挿入することができる特殊なブレークポイントが用意されています。

このブレークポイントを使ってプログラムを修正するために，まず***CTRL-F2***を押してプログラムを再ロードしてください。再ロードしたら***GRPH-B D***と入力して，その前のセッションで設定したブレークポイントをすべて解除してください。次に行96にカーソルを移

動して**F2**を押し，ブレークポイントを設定してください。それから**GRPH-V B**と押してBreakpointsウィンドウを開いてください。そして**GRPH-F10/S/E**と押し，プロンプトに対して*bufp*＋＋と入力してください。このように設定すると，プログラムの実行がこのブレークポイントに到達するたびに，式*bufp*＋＋が実行されます。ここで**F9**を押して，プログラムを実行してください。前と同じように次の2行を入力してください。

```
one two three
four five six
```

動作がかなり改善されたことに気がつくでしょう。総語数と総行数はまだ間違っているようですが，表は合っています。こんどは*printstatistics*ルーチンの先頭でプログラムの実行を停止して，このルーチンに渡される値が正しいかどうか見てみましょう。まず**CTRL-F2**と押してプログラムを再ロードしてください。次に行103にカーソルを移動し**F4**を押して，行103まで実行してください。実行したら引数*nlines*にカーソルを移動して**GRPH-F10 I**と押し，値を調べてください。2でなければならないのに6になっています。それでは，**main**の中で*printstatistics*ルーチンを呼び出している箇所まで戻って，そこの*nlines*の値を調べてみましょう。行35の*nlines*の下にカーソルを移動して**GRPH-F10 I**を押し，値を見てください。**main**の中の*nlines*の値は2で，これは合っています。その後の行をずっと見ていくと，行45で2つの引数*nwords*と*nlines*が逆になっていることに気づくでしょう。コンパイラには，引数の順番が逆になっていることはわかりません。

この2つのバグを修正すれば，プログラムは正しく動作するはずです。ファイルTCDEMO.EXEは，バグを入れる前のプログラムです。プログラムの正しい動作を見るには，このプログラムを起動してください。

Pascalのデバッグセッション

この章の残りの部分は，Pascalの例題プログラムを用いたデバッグセッションの説明です。C言語を使っている人は，前の節のTurbo Cの例題プログラムを用いたデバッグセッションを見てください。

間違いを探す

例題プログラムは，あらかじめコンパイルしてディスクの中に入っています。デバッグセッションを始める前に，バグのある例題プログラムを実行して，どこがおかしいか調べてみましょう。

例題プログラムを自分でコンパイルするときは，次のようにしてください。Turbo Pascalコンパイラのコマンドライン版TPC.EXEを使っている人は，次のように入力してリターンを押してください。

```
TPC /v TPDEMOB
```

統合環境版TURBO.EXEを使っている人は，Debug/Standalone debuggingコマンドの設定がオンになっていること，Compiler/Destinationコマンドの設定がDiskになっていることを確認してからコンパイルしてください。

例題プログラムを起動するために，次のようにプログラム名と3つのコマンドライン引数を入力して**リターン**を押してください。

```
TPDEMOB first second third
```

すると例題プログラムは，文を1行ずつ入力するように求めます。次のように2行入力してください。

```
ABC DEF GHI
abc def ghi
```

最後に何も入力しないで**リターン**を押して入力を終了させてください。TPDEMOBは，入力した文の分析結果を表示します。

```
9 letter(s) in 3 word(s) in 2 line(s)
  Average of 0.67 words per line

  Word length:  1    2    3    4    5    6    7    8    9    10
  Frequency:    0    0    0    0    0    0    0    0    0     0

  Letter:       M
  Frequency:    1    1    1    1    1    1    1    1    1    1    1    0    0    0    0
  Word starts:  1    0    0    1    0    0    1    0    0    0    0    0    0    0    0

  Letter:       Z
  Frequency:    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
  Word starts:  0    0    0    0    0    0    0    0    0    0    0    0    0    0    0

  Program name: A:¥TD¥TPDEMOB.EX
  Command line parameters: firs  secon  thrid
```

この出力には，次の5つの問題があります。

1. 語数が間違っている(6でなく3になっている)。
2. 1行当たりの語数が間違っている(3.00でなく0.67になっている)。
3. 文字の出現頻度表の各列の見出しが間違っている(2番目の表の最上行にはA..Mが，3番目の表の最上行にはN..Zが表示されなければならない)。
4. 文字の出現回数が間違っている(入力した2行にはどちらにもA..Iの文字が含まれているのに，表では各文字の出現回数が1回になっている)。
5. 各コマンドラインパラメータの最後の文字が失われ，でたらめな文字が表示されている。ただし最後のパラメータだけは正常である。

どの問題から取りかかるかを決める

まずやるべきことは，どの問題から取りかかるかを決めることです。そのための指針としては，〝最初〃に発生するように見える問題から取りかかることです。

このプログラムでは，手続き*Init*が呼び出されてデータを初期化した後，関数*GetLine*がキーボードからの入力を1行読み込み，手続き*ProcessLine*がそれを処理します。何も入力しないでリターンが押されるまで，これを繰り返します。手続き*ProcessLine*は，各文字列の長さと文字を調べ，カウンタ(グローバル変数)を更新します。次に手続き*ShowResults*が結果を表示します。最後に独立したサブプログラムにおいて，手続き*ParmsOnHeap*が，ヒープ領域にコマンドラインパラメータの線形リストを作成し，そのリストを表示します。

1行当たりの平均語数は，*ShowResults*により行数と語数を使って計算されます。先に見たように語数は間違っているので，*NumWords*が*ProcessLine*によりどのように更新される

かを調べてみる必要があるでしょう。しかし0.67という平均語数は，語数の間違いだけからは説明できません。これは*ShowResults*の計算にも間違いがあることを示唆します。したがって*ShowResults*も調べてみる必要があるでしょう。

表の各列の見出しは，*ShowResults*の要求に応じて表示されます。この見出しも間違っています。

前の節で説明した2番目と3番目のバグは，メインループが終了するまでは調べることができません。また，文字数と語数が間違っているので，手続き*ProcessLine*の内部に何か間違いがあることはかなり確かでしょう。したがってまず，手続き*ProcessLine*が，入力行を正しく処理するかどうか調べてみましょう。

プログラムの文字数と語数を数える部分を調べてバグを取り除いたら，*ParmsOnHeap*を調べて，最後(5番目)のバグを見つけ出して修正しましょう。

どの問題から取りかかるか大まかな計画が決まったので，いよいよデバッグを開始しましょう。

Turbo Debuggerを起動する

デバッグセッションを開始するために，次のように入力して**リターン**を押してください。

```
TD TPDEMOB first second third
```

するとTurbo Debuggerが起動して，バグのある例題プログラムをロードし，初期画面を表示します。このデバッグセッションから抜け出してDOSに戻りたいときは，**GRPH-X**を押すといつでも抜けられます。どこかに迷い込んでどうすることもできなくなったときは，**CTRL-F2**を押すと，例題プログラムを再ロードして最初からやり直すことができます（この方法でデバッグセッションを再開するときは，ブレークポイントやWatchesウィンドウ内の設定は解除されないことに注意してください)。

*ProcessLine*をデバッグするには，2つのやり方があります。ひとつは，ソースコードを1行ずつ実行しながら動作を調べる方法です。もうひとつは，*ProcessLine*の全体を一度に実行して入力を1行分処理し，その結果から動作を調べる方法です。文字数も語数も両方とも間違っているので，*ProcessLine*の内部で文字がどのように処理されるか注意深く調べる必要があるでしょう。

それでは，プログラムを実行して*ProcessLine*の中に入っていきましょう。それにはいくつかの方法があります。まず**F8**（Step overコマンド）を4回押してから**F7**（Trace intoコマンド）を1回押すことにより，*ProcessLine*に入ることができます。またカーソルを行230に移動して**F4**（Go to cursorコマンド）を押し，それから**F7**を押すこともできます。

その他にもいろいろな方法がありますが，ここでは次の方法を使いましょう。**GRPH-F9**を押してください。するとプロンプトボックスが現われて，どこまで実行するかコードアドレスを入力するように求めます。それに答えて*processline*と入力して**リターン**を押してください。するとプログラムが実行されて，*ProcessLine*に制御が移ります。プログラムの画面が現われて文字列の入力を求めてきたら，前と同じ文字列（ABC DEF GHI）を入力してください。

*ProcessLine*には，いくつかのループがあります。外側のループは，文字列をスキャンします。その内部には，非英字コードをスキップするループと，文字および単語を処理するもう1つのループがあります。カーソルを行132の**while**ループに移動して**F4**(Go to cursor)を押してください。

このループは，文字列の終わりに到達するまで，または文字を見つけるまで，入力行をスキャンします。英字かどうかは，論理関数*IsLetter*を呼び出して検査しています。**F7**を押して*IsLetter*に入ってください。*IsLetter*はネストされた関数で，引数として文字値をとり，その文字値が英字ならTrueを返し，英字でなければFalseを返します。*IsLetter*をちょっと見ただけで，大文字しか検出しないことがわかります。'A'..'Z'および'a'..'z'の範囲の文字コードを検出するようにするか，検査する前に小文字を大文字に変換するようにしなければなりません。

先に入力した2行の文字列を見てみると，バグの原因を推測するもう1つの手がかりが得られます。'A'から'I'までの大文字と小文字を入力しましたが，総文字数は入力した文字数の半分です。もうその理由はおわかりでしょう。

GRPH-F8(Until returnコマンド)を押して，*IsLetter*を呼び出した行まで戻ってください。2行目の入力abc def ghiは小文字だけしか含んでいないので，各文字は空白として扱われ，スキップされます。これが文字数と語数の間違いの原因です。また4番目のバグも説明できます。

ついでに説明すると，*IsLetter*の動作に間違いがあることを検証するための，もうひとつの強力な方法があります。**GRPH-D E**を押してEvaluateウィンドウを開き，次の式を入力してください。

```
IsLetter('a') = IsLetter('A')
```

'a'と'A'はどちらも英字ですが，この式の評価結果がFalseになることから，*IsLetter*では同等に扱っていないことが確認されます（EvaluateウィンドウやWatchesウィンドウを使うと，式を評価したり，代入したり，ここで行なったように関数や手続きを呼び出すことができます。詳しい説明は第6章を参照してください）。

Inspectコマンドで調べる

これで2つのバグが解決しました。まだ3つ残っています。2番目のバグは，先の2つのバグに比べて発見がずっと容易です。**GRPH-F8**を押して*ProcessLine*から出てください。行233にカーソルを移動し**F4**を押して，カーソルの位置まで実行してください。

TPDEMOBは，テキストを入力するように求めます。abc def ghiと入力して**リターン**を押してください。再びプロンプトが表示されたら**リターン**だけを押してください。ここで**F7**を押して*ShowResults*の中に入ってください。

これからやろうとしていることは，1行当たりの平均語数が間違っている理由を見つけ出すことです。*ShowResults*のソースコードを見てください。最初の行で算出されるのは，1行当たりの語数ではなく，1語当たりの行数です。この2つの項は，明らかに逆になっています。

これまでの経過から，変数*NumLines*と*NumWords*が期待どおりの(間違った)値を持つと信じてもいいでしょう。*NumLines*は2のはずです。また先に突き止めてまだ修正していない*IsLetter*のバグのために，*NumWords*は3のはずです。では*NumLines*と*NumWords*の値を調べてみましょう。*NumLines*にカーソルを移動して，**GRPH-F10 I**と押してください。するとInspectorウィンドウが開いて，*NumLines*のアドレス，データ型，現在の値(10進表記と16進表記の両方)が表示されます。*NumLines*の値は確かに2です。次に*NumWords*の値を調べて見ましょう。**ESC**を押してInspectorウィンドウを閉じ，*NumWords*にカーソルを移動してもう一度**GRPH-F10 I**を押してください(ショートカットの**CTRL-I**を使うこともできます)。有難いことに，*NumWords*は期待どおりの(間違った)値3です。これで次に進むことができます。

ちょっと待ってください。本当に次に進んでも大丈夫でしょうか。*ShowResults*の最初の行の計算には，プログラムの問題点のリストには入っていない，もうひとつの問題点があります。その問題というのは，割算を行なう前に，第2項(除数)が0であるかどうかをチェックしていないことです。もしプログラムを最初から実行して，データをまったく入力しなければ(プログラムが入力を求めたときに**リターン**を押す)，除数と被除数を逆にした後でも，プログラムはクラッシュするでしょう。

このことを確かめるために，**ESC**を押してInspectorウィンドウを閉じ，**GRPH-R P**と押してプログラムを再ロードし，**F9**を押して最初から実行し，プログラムの入力要求たいして**リターン**を押してください。プログラムの実行が終了し，エラーボックスに実行時エラーが表示されるはずです。

したがってこの文は，次のように修正する必要があります。

```
if NumLines <> 0 then
    AvgWords := NumWords / NumLines
else
    AvgWords := 0;
```

これで2番目と3番目のバグは解決します。

この機会に，Inspectorウィンドウを使って，このプログラムのデータの構造を調べてみましょう。行49の*LetterTable*の宣言のところにカーソルを戻してください。そして*LetterTable*のうえにカーソルを置いて，**GRPH-F10 I**と押してください。そうすれば，*LetterTable*がレコードの配列であること，要素数が26であることがわかるでしょう。カーソルキーを使って，配列の要素の表示をスクロールしてください。また**リターン**を押して，配列の各要素の内容を調べてください。この方法は，データ構造を調べるための非常に強力な方法です。この後で線形リスト*ParmsOnHeap*を調べるときに，その便利さがわかるでしょう。

Watchコマンドで調べる

こんどは3番目のバグ（表の列の見出しの間違いを引き起こす*ShowResults*の中のバグ）を探しましょう。前の節でゼロ除算エラーが発生することを確かめたときにプログラムの実行が終了しているので，**GRPH-R P**と押してプログラムを再ロードし，新たなセッションの準備をしてください。準備ができたら**GRPH-F9**を押し，*showresults*と入力して**リターン**を押してください。プログラムの画面が表示されたら，前と同じように，まずABC DEF GHIと入力して**リターン**を押し，次にabc def ghiと入力して**リターン**を押してください。最後に何も入力しないで**リターン**を押します。*ShowResults*でプログラムの実行が停止するはずです。

*ShowResults*は，ネストされた手続き*ShowLetterInfo*を使って，文字の出現頻度表を表示します。カーソルを行102に移動して**F4**を押してください。次に**F7**を押して*ShowLetterInfo*の中に入ってください。

*ShowLetterInfo*の中には3つの**for**ループがあります。最初のループは，列の見出しを表示します。2番目と3番目のループは，文字の出現回数を表示します。**F7**を使って行62の最初のループまで実行してください。それから*FromLet*にカーソルを置き**GRPH-F10 I**と押して，値を調べてください。同じようにして*ToLet*の値を調べてください。どちらの値も，正しいように見えます（*FromLet*の値は'A'，*ToLet*の値は'M'です）。**GRPH-F5**を押して，プログラムの画面表示がどうなっているか調べてください。調べ終わったら，どれかキーを押してModuleウィンドウに戻ってください。

このようなループを実行しながら調べるときは，Watchウィンドウが非常に便利です。

*ch*にカーソルを置いて**CTRL-W**を押してください。それでは，**F7**を使って**for**ループを1行ずつ実行してください。**F7**を押すと，期待どおり次の行(行63)の***Write***文に進みます。しかしWatchウィンドウを見ると，*ch*の値はすでに'M'になっています(このループは，すでに全回数実行されています)。キーワード**do**のすぐ右を見てください。セミコロン(；)が1つ余計についています。このセミコロンのために，この**for**ループは，何もしないで13回実行されます。そして次の行63の***Write***文に制御が移行しときに，*ch*の現在の値'M'が出力されるわけです。したがってこの余分のセミコロンを取り除けば，3番目のバグが解決されます。

残りのもうひとつのバグ

最後に，コマンドラインパラメータに関する不思議なバグを突き止めましょう。問題は，各コマンドラインパラメータ(最後のひとつを除く)の最後の文字が変化していることです。多分，文字列の長さを表わすバイトが間違っているか，文字列データが他のデータにより上書きされているかのどちらかでしょう。

GRPH-F9を押し，parmsonheapと入力して，**リターン**を押してください。*ParmsOnHeap*では，**for**文のループによりコマンドラインパラメータを始めから終わりまでスキャンする間に，ヒープ領域に線形リストを作成し，各文字列をコピーします。ポインタ*Head*はリストの先頭を指し示します。*Tail*はリストの最後のノードを指し示します。*Temp*は，新しいノードの領域を確保し初期化するための一時的な格納領域として使用されます。

ここではWatchウィンドウを使って，バグを見つけ出しましょう。文字列データがいつおかしくなるか調べるために，**CTRL-F7**を押して，次の式をWatchウィンドウに付け加えてください。

```
Tail^.Parm^
```

この式によって，リストの最後のノードに格納されている文字列データを監視することができます。もちろん*Tail*が行206で初期化されるまでは，この式の値は無意味です。また1行ずつ実行するかわりに，ループの各回の実行が停止したときにWatchウィンドウを見るようにしましょう。行207にカーソルを移動して**F2**を押し，ブレークポイントをセットしてください。次に**F9**を押して，ブレークポイントまで実行してください。DOS 3.xを使用しているときは，Watchウィンドウの中にTPDEMOB.EXEまでの全パスが表示されます(DOS 2.xを使用しているときは空文字列になります。この場合には，再度**F9**を押して先に進んでください)。文字列データは正しいように見えます。

F9を押して，ループをもう一度実行してください。今回も，文字列データは正しいよう

に見えます。これで，文字列がヒープ領域に，正しくコピーされていることがわかりました。それではInspectorウィンドウを使って，文字列がすでに壊れているかどうか調べてみましょう。行202の*Head*の上にカーソルを移動して，**GRPH-F10 I**と押してください。

リターンを押して，*Parm*が指し示す値を見てください。いま見ているのは，リストの第1のノードに対応する文字列データです。この文字列データは，すでに壊れています。**ESC**を押して，Inspectorウィンドウを閉じてください。次に下矢印キーを1回押し，**リターン**を押してください。するとInspectorウィンドウが開いて，リストの第2のノードが表示されます。もう一度**リターン**を押して，このノードに対応する文字列データを調べてください。文字列データは壊れていません。この文字列データは，Watchウィンドウの中の*Tail^.Parm^*の値と同じです。

F7を使ってこのループの中を1行ずつ実行しながら，Watchウィンドウを注視していてください。行198の***GetMem***の呼び出しが容疑の対象です。この呼び出しの前には*Tail^.Parm^*の値は*second*です。ところがこの呼び出しの直後には，*Tail^.Parm^*の最後の文字が壊れます。

何が起きているのでしょうか。この**for**ループは，コマンドラインパラメータのために，まず第1のレコード，文字列データ，第2のレコード，文字列データの順に領域を割り当てます。行198の***GetMem***の呼び出しは，文字列と長さバイト(1バイト)を格納するのに十分な大きさの領域を割り当てなければなりませんが，文字列の長さ***Length***(*s*)に1を加えていないことがわかるでしょう。そのため，行199の文字列の代入は，文字列のコピーはうまくいきますが，そのコピー先に割り当てられているよりも1バイト多く使います。その結果，***New***(*Temp*)が呼び出されるときに割り当てられる次のレコードの最初のバイトにより，文字列の最後の文字が上書きされてしまいます。最後のパラメータだけは，その後に続く*ParmRec*がないため，壊されずに済みます。

これで，このプログラムの中のすべてのバグ（前のリスト中のもの）を，解決することができました。このプログラムのコードを調べると，さらにいくつかのバグが見つかるはずです。それらのバグには，便宜上2つのアスタリスク(＊＊)で目印が付けてあります。いま見つけ出したバグと，目印が付けられているバグを修正して，TPDEMOB.PASを再コンパイルし，実行してみてください。ファイルTPDEMO.EXEは，第3章「**チュートリアル**」で使われているバグのないプログラムです。TPDEMOB.PASが正しく動作することを確認するには，TPDEMO.EXEの動作と比べてみてください。

第14章

80386プロセッサシステムでの仮想デバッグ

Turbo Debuggerには，80386プロセッサを使用したシステムの威力をフルに利用することができる仮想デバッガが用意されています。仮想デバッグでは，デバッグするプログラムは，デバッガがロードされていないかのように640KBより下位の全アドレス空間を使うことができます（Turbo Debuggerは，1MBより上位の拡張メモリにロードされます）。

仮想デバッグでも，通常のデバッグのときとまったく同じようにデバッグできます。通常のデバッグと異なるのは，プログラムがデバッグされていないときとまったく同じアドレスにロードされ，実行されることです。仮想デバッグは，大きなプログラムをデバッグするときや，プログラムが通常のデバッグのときのようにメモリの上位にロードされると姿を隠してしまうバグを見つけ出すときに，非常に便利です。

また仮想デバッグでは，フルプロセッサスピードによる，任意のメモリロケーションやI/Oポートに対する読み書きをウォッチできます。そのためハードウェアデバッガに余分なコストをかけなくても，ハードウェアデバッガを使用するときと同じ強力な機能を利用することができます。

仮想デバッグのために必要なもの

仮想デバッガを使うには，CPUに80386プロセッサを使用したコンピュータが必要です。また700KBの拡張メモリが必要です。拡張メモリの全部をRAMディスク，キャッシュ，その他のために使っている場合には，仮想デバッガを使いたいときに拡張メモリを解放する特別なCONFIG.SYSまたはAUTOEXEC.BATファイルを作成すると便利です。

仮想デバッガのデバイスドライバのインストール

仮想デバッガを使うには，システムの起動時に，仮想デバッガのデバイスドライバをシステムに追加しなければなりません。そのためにCONFIG.SYSファイルの中に次の行を挿入してください。

```
DEVICE = TDH386.SYS
```

TDH386.SYSをルートディレクトリ以外のどこかに置くときは，デバイスドライバのファイル名の一部として，ディレクトリパスを指定してください。

仮想デバッガは通常，DOSの環境変数のために256バイトを確保します。これだけでは足りないとき，またはそんなに必要とせずメモリを節約したいときは，-eオプションを使って，環境バイト数を指定してください。たとえば

```
DEVICE = TDH386.SYS -e2000
```

と指定すると，DOSの環境変数のために2000バイトが確保されます。

仮想デバッガの起動

仮想デバッガを起動するには，Turbo Debuggerを起動するときとまったく同じように，次のように入力して**リターン**を押してください。

```
TD386 [options] program [programoptions]
```

言い換えると，TDのかわりにTD386と入力するだけです。TD386は，Turbo Debuggerの実行可能プログラムを見つけ出して，拡張メモリにロードします。

拡張メモリを使う他のプログラムやデバイスドライバ(RAMディスクやキャッシュなど)があるときは，それらのプログラムのために確保すべき拡張メモリの大きさを指定しなければなりません。**-e**または**-t**オプションを使って，他のプログラムが使用する拡張メモリの大きさをKB単位で指定してください。たとえば次のように指定します。

```
TD386 -e512 myprog
```

このコマンドラインは，拡張メモリの最初の512Kを，他のプログラムのために確保したいことをTD386に知らせます。

おそらく他のプログラムのためにも同じ大きさの拡張メモリを確保することが多いでしょうから，設定を変更するまで指定した大きさの拡張メモリを確保することができる方法が用意されています。**-e**オプションと一緒に**-w**オプションを指定すると，**-e**オプションで指定した値を，TD386の実行可能なファイルの中に書き込むことができます。

-wオプションを指定すると，設定を書き込みたいTD386の実行可能なファイルの名前を入力するように求めます。DOS 3.0またはそれ以降のバージョンを使用しているときは，いま起動したTD386のパスとファイル名をプロンプトに表示します。**リターン**を押してこの名前を受け入れることもできますし，別のファイル名を入力することもできます。存在しないファイル名を入力したときは，TD386はその名前で新しい実行可能なファイルを作成します。

DOSのバージョン2.xを使用しているときは，設定を書き込みたいTD386プログラムのパス名とファイル名を入力しなければなりません。

TD386.EXEのコマンドラインオプションは次のとおりです。

-e####　他のプログラムにより使用される，拡張メモリ(EMS)の大きさ(KB数)を指定します。拡張メモリの低位から指定のサイズを，他のプログラム用の領域とします。

-w　　-**e**または-**f**で指定した新しいデフォルト値により，TD386.EXEを変更します。

-f####　(拡張メモリの中で)ページングによるEMSエミュレーションを有効にし，フレームセグメントを####(16進数)に設定します。最後の3文字は000でなければなりません(C000やE000のように)。このオプションは，Turbo DebuggerのEMS呼び出しに対してだけ有効です。

-f-　　EMSエミュレーションを無効にします(-**w**と-**f**による設定をオーバーライドします)。

-b　　割り込みが禁止されている(CLI)場合にも***CTRL-STOP***(***Ctrl-Break***)によるブレークを有効にします。

-t####　他のプログラムによって使用される，拡張メモリ(EMS)の大きさをKB単位で指定します。拡張メモリの高位から指定のサイズを，他のプログラム用の領域とします。これはPC-9801のMS-DOS 3.3のRAMDISK.SYSに対応するために用意されたオプションです。

通常デバッグと仮想デバッグの相違

Turbo Debuggerのほとんどの動作は，80386仮想デバッグ機能を使っているときも，通常デバッグのときとまったく同じです。次の点が相違します。

■***F10***/File/OS shellコマンドを使ってDOSのコマンドを実行するときに，デバッグしているプログラムはディスクに退避されません。このことは，DOSのプロンプトから他のプログラムを実行するのに，メモリが不足する場合があり得ることを意味します。

■プロテクトモードの命令**CLTS, LMSW, LTR, LGDT, LIDT, LLDT**を除く，ほとんどすべての80386の命令を含むプログラムをデバッグできます。

■仮想デバッグを行なっているときは，80386のすべての拡張アドレッシングモードと32ビットレジスタを使用することができますが，1Mbより上位のメモリにアクセスすることはできません。もしアクセスしようとすると，例外割り込みが生成されて，Turbo Debuggerに制御が移ります。

■80386プロセッサの仮想モードやプロテクトモードを使用するプログラムやデバイスドライバが動作しているときは，仮想デバッグは使えません。このようなプログラムやデバイスドライバには次のものがあります。

- DesqViewオペレーティングシステム
- Windows-386オペレーティング環境
- CEMM.SYSコンパックEMSシミュレータ
- 386^MAX

上記のプログラムや類似のプログラムを使用しているときは，TD386を起動する前に，それらのプログラムを終了させてください。

TD386のエラーメッセージ

TD386は，起動できないときは，次のようなエラーメッセージを表示して，DOSに戻ります。

TD386 error: 80386 device driver missing or wrong version

80386のデバイスドライバが見つからないか，バージョンが合っていない

DOSのコマンドラインからTD386を呼び出す前に，CONFIG.SYSファイルの中にTDH386.SYSデバイスドライバをインストールしなければなりません。

TD386 error: Can't enable A20 address line

A20アドレスラインを有効にすることができない

TD386が1メガバイトより上位のメモリにアクセスできません。このエラーは，システムがIBM完全互換でないときに発生することがあります。PC-9801では発生しません。

TD386 error: Can't find TD.EXE

TD.EXEが見つからない

TD386がTD.EXEを見つけられませんでした。

TD386 error: Couldn't execute TD.EXE

TD.EXEを実行できない

TD386がTD.EXEを実行できませんでした。

TD386 error: Environment too long; use -e#### switch with TDH386.SYS

環境変数が長すぎる．TD386.SYSに-e####スイッチを指定してください

THD386.SYSの後に，-eオプションを使って環境バイト数を変更する必要があります。

TD386 error: Not enough Extended Memory available

利用できる拡張メモリが足りない

メモリが足りません。システムに拡張メモリを増設するか，他の目的に使っているメモリを解放してください（たとえばRAMディスクを縮小する）。

TD386 error: Wrong CPU type(not an 80386)

CPUのタイプが違う（80386ではありません）

システムのCPUが，80386プロセッサではありません。

TD386を-**w**オプションで変更したときは，次のエラーが発生することがあります。

TD386 error: Cannot open program file

プログラムファイルをオープンできない

TD386 error: Cannot read program file

プログラムファイルを読み出すことができない

TD386 error: Cannot write program file

プログラムファイルを書き込むことができない

TD386 error: Program file corrupted or wrong version

プログラムファイルが壊れているか，バージョンが合っていない

TDH386.SYSエラーメッセージ

TDH386.SYSドライバに関して発生する可能性のあるエラーメッセージは次の2つです。

Wrong CPU type: TDH driver not installed
CPUのタイプが合っていない：TDHドライバはインストールされませんでした

Invalid command line: TDH driver not installed
コマンドラインの指定に間違いがある：TDHドライバはインストールされませんでした

付録A

コマンドラインオプション

DOSのコマンドラインからTurbo Debuggerを起動するとき，同時にコマンドラインオプションを使って動作を設定することができます。コマンドラインオプションを指定するための一般的な書式は，次のとおりです。

```
td  [options]  [program_name [program_args]]
```

カッコで囲まれている項は省略することができます。オプションの後にハイフン (-) を付けると，コンフィギュレーションファイルの中でそのオプションがオンに設定されていても，明示的にそれをオフにすることができます。

オプション	機能
-do	デバッガの画面を第2のディスプレイに表示する（IBM PCのみ)。
-dp	ディスプレイのページを切り替えることにより，デバッガの画面とプログラムの画面を切り替える。
-ds	デバッガの画面とプログラムの画面をスワップする。
-g，-g＋	デバッガの46行画面を使用可能にする（PC-9801)。
-g-	46行画面を使用不可にする（PC-9801)。
-g＜＃＞	46行画面に使うGVRAMのバンク（0または1）を指定する（PC-9801)。
-h，-?	ヘルプ画面を表示する。
-i	プロセスIDを切り替える。
-l	アセンブラモードで起動する。
-m＜＃＞	ヒープサイズ(KB)を設定する。 最大値：64K　　デフォルト：40K
-r	リモートシステム上でのデバッグをオンにする。
-rp＜＃＞	リモートリンクのためのCOMポートを設定する。 1：COM1(デフォルト)　2：COM2　3：COM3（PC-9801のみ)
-rs＜＃＞	通信速度を設定する。 ［PC-9801］ 1：9600bps　2：40/31Kbps（デフォルト) ［IBM PC］ 1：9600bps　2：40Kbps　3：115Kbps（デフォルト)
-sd＜*dir*＞	ソースファイルを探すディレクトリを指定する。
-sc	文字の大小を無視する。
-vn	43/50行の表示モードを使用不可にする（IBM PC)。
-vn-	43/50行の表示モードを使用可能にする（IBM PC)。
-vg	プログラム画面の退避時にグラフィックスイメージをそのまま保存する(IBM PC)。

付録B

Turbo Debuggerのユーティリティ

Turbo Debuggerのマスタディスクには，複数のユーティリティプログラムが一緒に入っています。

1つ目は，マイクロソフト社のコンパイラによって作成されたプログラムを，Turbo Debuggerでデバッグできるように変換するためのユーティリティです。

2つ目は，リモートデバッグを行なっているときに，リモートシステムに対して基本的なファイルメンテナンスコマンドを発行できるようにするユーティリティです。

3つ目は，プログラムからデバッグ情報（“シンボルテーブル”）を削除するユーティリティです。

その他にも，MAPファイルからデバッグ情報を読み出してプログラムファイルに連結するユーティリティ，デバッグ情報をダンプするユーティリティ，デバッグ情報を圧縮するユーティリティなどがあります。

TDUMPは，オブジェクトモジュールを逆アセンブルする（ここで言う逆アセンブルとは広い意味であり，機械語コードをニーモニック表現するという意味ではありません）プログラムです。

シンボルテーブルコンバータ：(CodeView→Turbo Debugger)

TDCONVRT.EXEユーティリティプログラムは，Microsoft Linkでリンクされたプログラムを，Turbo Debuggerでデバッグできる形式に変換します。このユーティリティで変換するプログラムは，CodeViewでデバッグするときとまったく同じやり方で準備しなければなりません。

DOSから起動する

形式変換したいプログラムをコンパイルしリンクしたら，DOSのコマンドラインに次のように入力してください。

```
TDCONVRT oldname [newname]
```

*oldname*は，Microsoft Linkで作成した実行可能プログラムファイルの名前です。*newname*は，変換後のTurbo Debuggerでデバッグ可能なファイルのための新しいファイル名です。*newname*を指定しないときは，変換されたファイルのファイル名は，元のファイル名*oldname*と同じになります。

このユーティリティは，プログラムがCodeViewのバージョン1またはバージョン2のどちらの形式で作成されているかを自動的に検出し，それに適合した処理をします。

エラーメッセージ

この変換ユーティリティは，CodeView形式のプログラムをTurbo Debuggerの形式に変換できないときは，その問題に応じて次のようなエラーメッセージを表示します。

Can't convert CodeView version 1 symbol tables
CodeViewバージョン1のシンボルテーブルは変換できません。

CodeViewのバージョン1にしか使えないシンボルテーブルを持つプログラムを，変換しようとしました。CodeViewのバージョン2用のシンボルテーブルを生成する，もっと新しいバージョンのマイクロソフト社のツールを使って，再コンパイルし再リンクしなければなりません。

Can't creat file: ***fname***

ファイルを作成できません。

TDCONVRTが，ファイル*fname*を作成することができませんでした。ディスクに十分な空き領域がない可能性があります。不要なファイルをいくつか削除して，もう一度やってみてください。

Can't translate a packed symbol table

圧縮されたシンボルテーブルは変換できません。

CVPACKユーティリティで圧縮されたシンボルテーブルを変換しようとしています。TDCONVRTは，圧縮されていないシンボルテーブルしか変換できません。プログラムを再リンクし，CVPACKユーティリティを使わないで，TDCONVRTで変換してください。

Error reading file: ***fname***

ファイル*fname*の読み込み中にエラーが発生しました。

このエラーは，通常，ファイルが書き込まれているディスクセクタが読めなくなったときに発生します。変換しようとするファイルを再リンクしてディスクに書き込み，もう一度変換してみてください。

Error writing file: ***fname***

ファイル*fname*の書き込み中にエラーが発生しました。

このエラーは，通常，ディスクが一杯になったことを示します。ファイルをいくつか削除して，もう一度変換してみてください。

File not found: ***fname***

ファイル*fname*が見つかりません。

TDCONVERTは，指定されたファイルを見つけることができませんでした。ファイル名に拡張子を指定しなかったときは，TDCONVRTは.EXEが指定されたものと見なします。

Not enough memory to perform conversion

変換を行なうのにメモリが足りません。

シンボルテーブルをTurbo Debuggerの形式に変換しているときに，メモリが足りなくなりました。ロードしている常駐ユーティリティをメモリからはずして，もう一度変

換してみてください。それでもメモリが足りないときは，プログラムのデバッグ情報の量を減らしてみてください。プログラムのデバッグ情報の量を減らすには，いくつかのモジュールだけをデバッグ情報の生成をオンにして，コンパイルまたはアセンブルします。

Program does not have a valid symbol table

プログラムに適正なシンボルテーブルがありません。

MS-LINKにより生成された適正なシンボルテーブルを持たないプログラムを変換しようとしました。TDCONVRTはMS-LINKにより生成されたシンボルテーブルしか変換できません。

Symbol table contains untranslated fields

シンボルテーブルに変換できないフィールドが含まれています。

変換しようとしているプログラムのシンボルテーブルに，TDCONVRTが変換できないフィールドが含まれています。TDCONVRTは，マイクロソフト社のCやアセンブラで書かれたプログラムは問題なく変換できますが，同社のBASICやPascalで書かれたプログラムを変換するときは問題が起きる可能性があります。

Symbol table has invalid format

シンボルテーブルの形式が正しくありません。

シンボルテーブルが壊れているか，MS-LINKの古いバージョンで生成されたものである可能性があります。MS-LINKの最新バージョンを使って再リンクし，新しい.EXEファイルとシンボルテーブルを生成してください。

TDRFユーティリティ

TDRF (Remote File transfer) ユーティリティは，リモートシステム上のTDREMOTEユーティリティと協力して動作します。TDRFを使うと，リモートシステムに対してほとんどのDOSのファイルメンテナンス操作を行なうことができます。次のような操作ができます。

- ■リモートシステムへのファイルのコピー
- ■リモートシステムからのファイルのコピー
- ■ディレクトリの作成
- ■ディレクトリの削除
- ■ディレクトリの内容の表示
- ■ディレクトリの変更
- ■ファイル名の変更
- ■ファイルの削除

リモートシステム上でTDREMOTEを起動する方法とリモートデバッグの説明は，付録G「**リモートデバッグ**」を参照してください。

リモートシステム上でTDREMOTEを起動すると，いつでもTDRFを使うことができます。TDRFは，DOSのプロンプトから直接起動することもできますし，Turbo Debuggerの内部から***F10***/File/OS shellコマンドによりDOSにアクセスして起動することもできます。DOSの内部から起動するときは，リモートシステム上のプログラムをデバッグしているときにも起動できます。この機能は，デバッグしているプログラムが必要とするファイルを，リモートシステムに移しておくのを忘れたときなどに便利です。

以下のTDRFの説明では，キーボードから入力しているシステムを“ローカルシステム”と呼び，ローカルシステムのファイルを“ローカルファイル”と呼びます。またケーブルで接続されたもうひとつのシステムを“リモートシステム”と呼び，リモートシステムに置かれているファイルを“リモートファイル”と呼びます。

DOSのコマンドラインからTDRFを起動する

TDRFを起動するためのコマンドラインの一般的な書式は，次のとおりです。

```
TRDF [options] command [arguments]
```

*options*は，リモートリンクの通信速度と使用するポートを指定します。

*command*は，行なおうとする操作を指定します。COPY, DEL, MDのようにDOSコマンドラインで入力しなれたやり方で指定することもできますし，1文字の省略形で指定することもできます。

たとえば，リモートシステムのカレントディレクトリの中のABCで始まるすべてのファイルを表示させるには，次のように指定します。

```
TDRF DIR ABC*
```

次に，指定できるすべてのオプションとコマンドについて説明します。

TDRFのコマンドライン

オプションは，頭にハイフン(-)またはスラッシュ(/)を付けなければなりません。TDRFユーティリティに対して指定できるコマンドラインオプションは次のとおりです。

-rs*N*　リモートリンクの通信速度を指定します。
-rsオプションは，リモートリンクの通信速度を指定します。リモートシステム上でTDREMOTEを起動するときに指定したのと同じ速度を指定しなければなりません。*N*は1，2または3です。1は9600bps，2は40Kbps，3は115Kbpsを示します。*N*が大きいほどデータ転送速度が速くなります。TDRFのデフォルトは通常**-rs3**(最高速度)に設定されています。

-rp*N*　リモートリンクのポートを指定します。
-rpオプションは，リモートリンクのために使用するポートを指定します。*N*は1または2です。1はCOM1を示し，2はCOM2を示します。

-w　コマンドラインオプションを実行可能なプログラムファイルに書き込みます。
-wを使うと，TDRFのコマンドラインオプションを，ディスク上のTDRFの実行可能なプログラムのイメージに書き込むことができます。書き込んだ設定が新しいデフォルト値になります。**-w**オプションを指定すると，コマンドラインオプションを書き込みたいプログラムの名前を入力するように求めます。
DOS 3.0またはそれ以降のバージョンを使用しているときは，いま起動したTDRFのパスとファイル名をプロンプトに表示します。**リターン**を押してこの名前を受

け入れることもできますし，別の実行可能なファイルの名前を入力することもできます。入力されたファイル名が見つからないときは，TDRFはその名前で新しいファイル(TDRFのコピー)を作成します。
DOSのバージョン2.xを使用しているときは，いつでも設定を書き込みたいプログラムのパス名とファイル名を入力しなければなりません。

TDRFのコマンド

TDRFユーティリティがサポートするコマンドの名前と機能は次のとおりです。

COPY ローカルシステムからリモートシステムにファイルをコピーします。COPYのかわりにCOPYTOと入力することもできます。このコマンドの1文字の省略形はTです。
COPYコマンドの後にファイル名を1つ指定すると，そのファイルがローカルシステムからリモートシステムのカレントディレクトリにコピーされます。
ローカルシステムのファイル名の後にもう1つファイル名を指定すると，そのローカルファイルが，リモートシステムの指定したコピー先にコピーされます。たとえば次のように入力すると，

```
TDRF COPY TEST1 ¥MYDIR
```

ローカルシステムのファイルTEST1が，リモートシステムのファイル¥MYDIR¥TEST1にコピーされます。
ファイルの指定に，ワイルドカード＊と？を使うこともできます。たとえば次のように入力すると，

```
TDRF T TC*.* A:¥TCDEMO
```

ローカルシステムのカレントディレクトリの中のTCで始まるすべてのファイルが，リモートシステムのドライブAのサブディレクトリ¥TCDEMOにコピーされます。

COPYFROM リモートシステムからローカルシステムにファイルをコピーします。このコマンドの1文字の省略形はFです。
COPYFROMコマンドの後にファイル名を1つ指定すると，そのファイルが，リモートシステムからローカルシステムのカレントディレクトリにコピーされます。

リモートシステムのファイル名の後にもう1つファイル名を指定すると，そのリモートファイルが，ローカルシステムの指定したコピー先にコピーされます。たとえば次のように入力すると，

```
TDRF COPYFROM MYFILE ..
```

リモートシステムのファイルMYFILEが，ローカルシステムのカレントディレクトリの親ディレクトリにコピーされます。
ファイルの指定に，ワイルドカード＊と？を使うこともできます。

DEL リモートシステムのファイルを削除します。このコマンドの1文字の省略形はEです。
ドライブ名もディレクトリも指定しないで，ファイル名だけを指定したときは，リモートシステムのカレントディレクトリから，そのファイルが削除されます。たとえば次のように入力すると，

```
TDRF DEL ¥XYZ
```

リモートシステムのルートディレクトリからファイルXYZが削除されます。
ファイルの指定に，ワイルドカード＊と？を使うこともできます。

DIR リモートシステムのディレクトリの内容を一覧表示します。このコマンドの1文字の省略形はDです。
このコマンドはDOSのDIRコマンドと非常によく似ています。ファイル名に対してマッチング条件を指定しないときは，指定したディレクトリのすべてのファイル名が表示されます。ワイルドカードを使ってマッチング条件を指定したときは，条件に合致するファイル名だけが表示されます。***STOP***キーを押すと，ディレクトリ表示を中断することができます。
ディレクトリの表示形式は，DOSのDIRコマンドの表示形式とまったく同じです。たとえば次のように入力すると，

```
TDRF DIR ¥SYS¥*.SYS
```

次のように表示されます。

```
Directory of  A:¥SYS
          .              <DIR>      88-09-08   8:01
          ..             <DIR>      88-09-08   8:01
          EMSDRIVE SYS     31934    88-07-13   0:00
          GRAPH    SYS     34671    88-07-13   0:00
```

REN リモートシステムのファイルの名前を変更します。このコマンドの1文字の省略形はRです。

このコマンドには,2つのファイル名(元のファイル名と新しいファイル名)を指定しなければなりません。新しいファイル名には，ファイル名の一部として，他のディレクトリを指定することもできます。ただし他のドライブは指定できません。たとえば次のように入力すると,

```
TDRF REN TEST1 ¥TEST2
```

リモートシステムのカレントディレクトリの中のファイルTEST1を,ルートディレクトリの中のファイルTEST2に変更します。この機能によって，あるディレクトリのファイルを別のディレクトリに移動することができます。2つのファイル名に同じディレクトリを指定すると，ファイルの名前だけを変更することになります。

MD リモートシステムに新しいディレクトリを作成します。このコマンドの1文字の省略形はMです。

コマンドの後に，作成するディレクトリの名前を指定しなければなりません。ディレクトリパスを指定しないときは，新しいディレクトリは，リモートシステムのカレントディレクトリの中に作られます。たとえば次のように入力すると,

```
TDRF MD TEST
```

リモートシステムのカレントディレクトリの中に,TESTという名前のディレクトリが作成されます。

RD リモートシステムの既存のディレクトリを削除します。このコマンドの1文字の省略形はKです。

コマンドの後に，削除するディレクトリの名前を指定しなければなりません。ディレクトリパスを指定しないときは，リモートシステムのカレントディレクトリの中のディレクトリが削除されます。次のように入力すると,

```
TDRF RD MYDIR
```

リモートシステムのカレントディレクトリから，MYDIRという名前のディレクトリが削除されます。

CD リモートシステムのカレントディレクトリを，別のディレクトリに変更します。このコマンドの1文字の省略形はCです。
コマンドの後に，移行するディレクトリ名を指定しなければなりません。ドライブ名だけ，またはドライブ名とディレクトリ名を同時に指定することもできます。次のように入力すると，

```
TDRF CD A:ABC
```

ドライブAがリモートシステムのカレントドライブになり，ディレクトリABCがカレントディレクトリになります。

TDRFのエラーメッセージ

TDRFユーティリティを使っているときに表示される可能性のあるエラーメッセージは次のとおりです。

Can't create file on local system: ***fname***
ローカルシステムにファイル*fname*を作成できません。

COPYFROMコマンドを使ってリモートシステムからファイル*fname*をコピーするときに，ローカルシステムにファイル*fname*を作成できませんでした。このエラーは，ローカルシステムのディスクが一杯のとき，またはファイルの名前がローカルシステムのディレクトリの名前と同じであるときに発生します。

Can't modify exe file
実行可能ファイルを変更できません。

コマンドラインオプションを書き込むために指定したファイルが，TDRFユーティリティの適正なコピーではありません。**-w**オプションで変更できるのは，TDRFユーティリティのコピーだけです。

Can't open exe file to modify

変更する実行可能ファイルがオープンできません。

コマンドラインオプションを書き込むファイルがオープンできません。不正なファイル名を入力した可能性があります。

Error opening file: ***fname***

ファイル*fname*をオープン中にエラーが発生しました。

リモートシステムに転送するファイル*fname*がオープンできませんでした。指定した名前のファイルが存在しないか，不正なファイル名を入力した可能性があります。

Error writing file: ***fname***

ファイル*fname*を書き込み中にエラーが発生しました。

ローカルシステムのディスクにファイル*fname*を書き込んでいるときに，エラーが発生しました。このエラーは，通常，ディスクが一杯になったときに発生します。ディスクのファイルをいくつか削除して，リモートシステムからコピーしたいファイルを書き込むためのスペースを空けてください。

Error writing file ***fname*** **on remote system**

リモートシステムにファイル*fname*を書き込み中にエラーが発生しました。

リモートシステムのディスクにファイル*fname*を書き込んでいるときに，エラーが発生しました。このエラーは，通常，リモートディスクが一杯になったときに発生します。リモートディスクのファイルをいくつか削除して，転送したいファイルを書き込むためのスペースを空けてください。

File name is a directory on remote

ファイル名と同名のディレクトリがリモートシステムに存在します。

ローカルシステムからリモートシステムにファイルをコピーしようとしましたが，リモートシステムにそのファイルと同名のディレクトリが存在します。COPYコマンドの2番目の引数に別のファイル名を指定してください。

Interrupted

中断されました。

リモートシステムとの間に通信が確立するのを待っている間に，***STOP***キーが押されました。

Invalid command: ***cmd***

コマンドが無効です。

TDRFが識別できないコマンドcmdを入力しました。各コマンドに対して，DOS形式のコマンド名を使うこともできますし，1文字の省略形を使うこともできます。

Invalid command line option: ***opt***

コマンドラインオプションが無効です。

DOSのコマンドラインからTDRFを起動するときに，不正なコマンドラインオプションを指定しました。

Invalid destination disk drive

目的ディスクドライブが無効です。

コマンドの引数の中に，存在しないディスクドライブを指定しました。リモートシステムのドライブの数は，ローカルシステムのドライブの数と同じとは限りませんので，注意してください。

No matching files on remote

リモートシステムにマッチするファイルがありません。

DIRコマンドを発行しましたが，リモートシステムの指定したディレクトリに，ファイルがひとつも存在しないか，マッチング条件にマッチするファイルがありません。

No remote command specified

リモートコマンドが指定されていません。

TDRFの後にコマンドを何も指定しませんでした。TDRFは何もすることがありません。

Too few arguments

引数が足りません。

コマンドに必要な引数を指定しませんでした。

Too many arguments

引数が多すぎます。

コマンドに引数を余計に指定しました。引数は2つまでです。3つまたはそれ以上の引数を必要とするコマンドはありません。また引数が1つだけのコマンドもあります。

Wrong version of TDREMOTE

TDREMOTEのバージョンが合っていません。

TDRFとTDREMOTEの一緒に使用できないバージョンを使用しています。各ユーティリティの最新バージョンを使用してください。

シンボルテーブル削除ユーティリティ：TDSTRIP

TDSTRIPユーティリティは，TLINKにより/vオプションを指定して生成した実行可能なプログラムから，シンボルテーブルを削除します。/vオプションを指定しないで再リンクするよりも，このユーティリティを使って削除する方が簡単です。またこのユーティリティを使って，実行可能なプログラムからシンボルテーブルを分離し，別のファイルに入れることもできます。これは.EXE形式を.COM形式に変換し，デバッグ用シンボルテーブルはそのまま残しておきたいときに便利です。このユーティリティは，シンボルテーブルを入れたファイルに.TDSという拡張子を付けます。Turbo Debuggerは，シンボルテーブルの付いていないプログラムをロードするときは，この拡張子の付いているファイルを探します。

TDSTRIPのコマンドライン

TDSTRIPを起動するときのDOSのコマンドラインの一般的な書式は，次のとおりです。

```
TDSTRIP [options] exename [outputname]
```

*options*には，**-s**オプションまたは**-c**オプションを指定することができます。

-sオプションを指定しないときは，.EXEファイル*exename*からシンボルテーブルが削除されます。出力ファイル*outputname*を指定すると，元の.EXEファイルはそのまま残され，シンボルテーブルを削除したファイルがファイル名*outputname*として作成されます。

-sオプションを指定したときは，.EXEファイル*exename*から削除されたシンボルテーブルは，.EXEファイル*exename*と名前が同じで，拡張子が.TDSのファイルに入れられます。出力ファイル*outputname*を指定したときは，シンボルテーブルは*outputname*に入れられます。

*exename*に拡張子を指定しないときは，.EXEを指定したものとみなされます。*outputname*に拡張子を指定しない場合には，**-s**を指定しないときは.EXEが付け加えられ，**-s**を指定したときは.TDSが付け加えられます。

次にTDSTRIPのコマンドラインの例をいくつか示します。

```
TDSTRIP MYPROG
```

MYPROG.EXEからシンボルテーブルが削除されます。

```
TDSTRIP -s MYPROG.OLD
```

MYPROG.OLDからシンボルテーブルが削除され，削除されたシンボルテーブルはMYPROG.TDSに入れられます。

```
TDSTRIP MYPROG MYPROG.NEW
```

MYPROG.EXEはそのまま残され，MYPROG.NEWという名前でMYPROG.EXEのシンボルテーブルのないコピーが作成されます。

```
TDSTRIP -s MYPROG MYSYMS
```

MYPROG.EXE からシンボルテーブルが削除され，削除されたシンボルテーブルがMYSYMS.TDSに入れられます。

-**c**オプションを指定すると，入力.EXEファイルが.COMファイルに変換されます。-**c**オプションと-**s**オプションを一緒に指定すると，シンボルテーブルの付いた.EXEファイルを.COMファイルと.TDSシンボルファイルに変換することができます。この変換を使うと，Turbo Debugger により，すべてのデバッグ情報を利用しながら.COMファイルをデバッグすることが可能になります。

ただし，すべての.EXEファイルが.COMファイルに変換できるわけではありません。開始アドレスが100Hであること，セグメントの値が一定であること，その他の条件が必要です。この制限は，TLINKの/tオプションを使うときの制限と同じです。

TDSTRIPのエラーメッセージ

TDSTRIPユーティリティを使っているときに表示される可能性のあるエラーメッセージは，次のとおりです。

Can't creat file: ***fname***

ファイル*fname*を作成できません。

TDSTRIPが出力ファイル(シンボルファイルまたは実行可能ファイル)***fname***を作成できませんでした。ディスクが一杯になったか，不正なファイル名を指定したかどちらかです。

Can't open file: ***fname***

ファイル*fname*をオープンできません。

TDSTRIPが，シンボルテーブルを削除する.EXEファイル*fname*を見つけられませんでした。

Error reading from input exe file

入力実行可能ファイルから読み込み中にエラーが発生しました。

入力実行可能ファイルから読み込んでいるときにエラーが発生しました。ディスクが読めなくなっている可能性があります。もう一度やってみてください。

Error writing to output file: ***fname*** **; disk may be full**

出力ファイル*fname*に書き込み中にエラーが発生しました。ディスクが一杯かもしれません。

出力ファイル(シンボルファイルまたは実行可能ファイル)*fname*に書き込むことができませんでした。このエラーは，通常，ディスクが一杯になったときに発生します。ファイルをいくつか削除して，TDSTRIPにより作成されるファイルのために空き領域を用意する必要があります。

Input file is not an .exe file

入力ファイルが実行可能ファイルではありません。

適正な実行可能ファイルでないファイルを入力ファイルに指定しました。シンボルテーブルを削除できるのは.EXEファイルだけです。なぜならTLINKがシンボルテーブルを付け加えることができるのは.EXEプログラムだけだからです。.COMファイル形式のプログラムは，シンボルテーブルを持ちません。したがってTDSTRIPで処理することはできません。

Invalid command line option: ***opt***

コマンドラインオプション*opt*が無効です。

DOSのコマンドラインからTDSTRIPを起動するときに，不正なコマンドラインオプションを指定しました。

Invalid exe file format

実行可能ファイルの形式が正しくありません。

入力ファイルは.EXE形式のプログラムファイルのように見えますが，何か問題があります。TLINKを使ってリンクをやり直してください。

Not enough memory

メモリが足りません。

TDSTRIPが.EXEファイルをロードして処理するのに十分なメモリがありません。このエラーは，非常に特殊な状況でしか発生しません(TDSTRIPは非常に小さなメモリしか必要としないからです)。システムをリブートしてからもう一度やってみてください。リブートしないと解放されないメモリ割り当てを行なうプログラムを使用していた可能性があるからです。

Program does not have a symbol table

プログラムにシンボルテーブルがありません。

適正な.EXE形式のファイルですが，シンボルテーブルがないファイルを指定しました。

Program does not have a valid symbol table

プログラムのシンボルテーブルは正しくありません。

.EXEファイルの末尾のシンボルテーブルが，適正なTLINKのシンボルテーブルではありません。このエラーは，.EXEファイルがTLINK以外のリンカによって生成されたプログラムであるときに発生します。

Too many arguments

引数が多すぎます。

TDSTRIPには，引数を2つまで指定できます。最初の引数は，実行可能なプログラムの名前です。2つ目の引数は，シンボルまたは実行可能なプログラムを書き込むための出力ファイルです。

You must supply an exe file name

.EXEファイル名が必要です。

TDSTRIPを起動するときに，シンボルテーブルを削除したい.EXEプログラムファイルの名前を指定しませんでした。

TDMAPユーティリティ

TDMAPユーティリティは，.MAPファイル(リンカによって生成された，プログラムのすべてのパブリックシンボルを含むASCII形式のファイル)を，Turbo Debuggerでデバッグできる形式で.EXEファイルに連結します。このユーティリティを使うと，他社のコンパイラとリンカでコンパイルされリンクされた実行可能なプログラムをデバッグすることができます。

TDMAPを使うための書式は次のとおりです。

```
TDMAP filename [/Esourceextension]
```

TDMAPは，*filename*.MAPからデバッグ情報を読み込み，それを*filename*.EXEに連結します。/Eは，デバッグ情報を連結する実行可能なプログラムファイルの拡張子を指します。プログラムファイルの拡張子が.EXEでないときは，このオプションにより拡張子を指定してください。

プログラムHELLO.PASにデバッグ情報のレコードを連結したいときは，次のように指定します。

```
TDMAP hello /Epas
```

TDPACKユーティリティ

TDPACKユーティリティは,.EXEファイルに連結されるデバッグ情報を圧縮して,実行可能なプログラムファイルのサイズを縮小します。このユーティリティは，文字列やデータ型に関する情報のような，重複した冗長な情報を取り除くことにより，デバッグ情報を圧縮します。

TDPACKを使うための書式は次のとおりです。

```
TDPACK filename
```

TDPACKは,拡張子が指定されないときは.EXEであるとみなします。また*filename*.EXEを見つけることができないときは,*filename*.COMと*filename*.TDSを探します(.TDSファイルは,Turbo Debuggerが.COMファイルをデバッグするときに使用するシンボルテーブルが入っているフィルです)。

TDUMPユーティリティ

TDUMPユーティリティは，一般的なモジュール逆アセンブラです。このユーティリティを使うと，プログラムファイルの構造を調べることができます。

TDUMPユーティリティは，可能な限りインテリジェントに，ファイルを構成要素に分解しようとします。そのために，まずファイルの拡張子を見ます。そして拡張子が識別できたら，その種類にしたがってファイルの構成要素を表示します。TDUMPが識別できるのは,.EXE,.OBJ,.LIBファイルです。これ以外の種類のファイルは，ファイルの内容をそのまま16進数で表示します。

TDUMPを使うと，ファイルの内部構造を調べることができます。TDUMPは，ただファイルの内容を表示するだけでなく，ファイルの構造も示します。またファイルの構造が拡張子と一致するかどうか確かめるので，TDUMPを使ってファイルが正常であるかどうかを調べることができます。

TDUMPのコマンドライン

TDUMPを起動するときのDOSのコマンドラインの一般的な書式は次のとおりです。

```
TDUMP [options] inputfile [outputfile]
```

*inputfile*は，内容を表示(または“ダンプ”)させたいファイルです。*outputfile*はTDUMPの出力を書き込みたいファイルの名前で，必要に応じて指定することができます（DOSのリダイレクション指示“>”を使うこともできます)。*options*は，次の節で説明するTDUMPのオプションを示します。

TDUMPのオプション

TDUMPに対しては，いくつかのオプションスイッチを指定できます。すべてのオプションは，頭にハイフン(-)を付けなければなりません（ハイフンのかわりにスラッシュ/を使うこともできます)。次の2つの例は同等です。

```
TDUMP -el -v demo.exe
TDUMP /el /v demo.exe
```

-aと-a7オプション

TDUMPは，ファイルの種類に応じて自動的に出力の表示形式を選択します（種類が識別できないファイルは，16進数で表示します）。ただし-**a**または-**a7**オプションを指定することにより，強制的にASCII形式で表示させることができます。

ASCII表示では，オフセットで位置を示しながら，ファイルの内容がASCIIコードの表示可能な文字で表示されます。表示できないキャラクタ（コントロールキャラクタなど）はピリオドで表示されます。またオフセットも表示されます。

-**a7**オプションは，ハイ-ASCIIキャラクタ（最上位のビットが1のASCIIコード）を対応するロウ-ASCIIキャラクタ（最上位のビットが0のASCIIコード）に変換します。それ以外の点は-**a**と同じです。このオプションは，ハイ-ASCIIキャラクタがフラグとして使用されているファイル(WordStarファイルなど）をダンプするときに便利です。

-e,-el,-erオプション

これらの3つのオプションを指定すると，ファイルを強制的に実行可能(.EXE)ファイルの表示形式で表示させることができます。

.EXEファイル表示では，表示の最上部にファイルの種々の情報が表示され，その下にグローバルシンボルテーブルとモジュールテーブルが表示されます。

-**el**オプションは，行番号の表示を抑制します。それ以外の点は-**e**オプションとまったく同じです。

-**er**オプションは，リロケーションテーブルの表示を抑制します。それ以外の点は-**e**オプションとまったく同じです。

-**elr**と指定すると，行番号とリロケーションテーブルの両方の表示を抑制することができます。

-hオプション

-**h**オプションを指定すると，ファイルを強制的に16進数形式で表示させることができます。16進数表示では，オフセットを示す数字，ファイルの内容を示す16進数，各16進数に対応するASCII文字が表示されます。対応する表示可能な文字がないときは，ピリオドが表

示されます。

TDUMPは，入力ファイルの拡張子が識別できないときは，16進数形式で表示します。ただしオプションにより別の表示形式が指定されているときは，指定されている表示形式で表示します。

-lオプション

-**l**を指定すると，ファイルを強制的にライブラリ(.LIB)ファイルの表示形式で表示させることができます。ライブラリファイルは，オブジェクトファイルを集めたものです（オブジェクトファイルについては，次の-**o**オプションの説明を見てください)。

ライブラリファイル表示では，まずそのライブラリに特有の情報が表示されます。次に各オブジェクトファイル名が表示され，その次に各オブジェクトファイルの各レコードが表示されます。

-oオプション

-**o**を指定すると，ファイルを強制的にオブジェクト(.OBJ)ファイルの表示形式で表示させることができます。オブジェクトファイルは，リンカにコマンドやデータを渡して.EXEファイルの生成の方法を知らせるためのコマンドレコードを含みます。

オブジェクトファイル表示では，各レコードとそれに付属するデータが，レコードごとに表示されます。

-vオプション

-**v**オプションは，どのファイル形式に対しても同じように，通常TDUMPが表示を読みやすくするためにダンプ表示の中に挿入するすべての説明を抑制します。そのため-**v**オプションを指定すると，文字どおりファイルの構成要素だけの非常に簡潔な表示になります。この表示を見て理解するには，上級プログラマでなければなりません。

付録C

技術ノート

この付録は，Turbo Debuggerの動作の基礎にある技術の詳細を理解したいと望む上級ユーザを対象としています。ここでの説明がまったく理解できなくても，自信をなくさないでください。デバッガを上手に使いこなすのに，ここに取り上げられている技術事項を理解する必要はありません。

この付録の中のデバッガとDOS，ハードウェア，プログラムとの相互作用の説明は，Turbo Debuggerのもとで実行されるプログラムの動作がどのように変化する可能性があるかを理解するのに役立つはずです。

またシステムがかなり簡単に壊れる理由と，それを避ける方法についても説明してあります。

ロードアドレスと空きメモリの変化

Turbo Debuggerは，メモリ内で自分がロードされている位置の後にプログラムをロードします。これにより2つの重要な結果が生じます。プログラムが通常より上位のセグメントアドレスにロードされることと，使用できる空きメモリが少なくなることです。

異なるアドレスにロードされ，異なるアドレスにアクセスされる結果，新たに現われるバグや，現われなくなるバグがあります。また空きメモリの大きさが減少することにより，メモリの割り当てやメモリ使用上のバグは，再現が難しいものもあります。

CPUに80386を使用したシステムでは，TD386仮想デバッグプログラムを使うことができます。TD386仮想デバッグプログラムを使うと，これらの問題を解消することができます。仮想デバッグの詳しい説明は第14章を参照してください。

システムの破壊

デバッガは，システムのメモリのすべてのアドレスに対して読み書きができます。そのため，DOSの内部や，メモリアドレスのゼロ番地から始まる割り込みテーブルなどの，プログラムの外部のメモリロケーションの内容を変更することにより，システムを壊す可能性があります。

たとえば，アドレス0000：0040のハードウェアクロック割り込みベクターを変更すると，ほとんど確実に問題が発生します。

DOS内部のトレースとプロセスIDの切り替え

Turbo Debuggerは，デバッグしているプログラムのファイルハンドルと干渉を引き起こさずにファイルをオープンしたりクローズしたりすることができるように，現在実行中のプロセスが自分自身であるかデバッグしているプログラムであるかを監視しています。そしてデバッガからプログラムが起動されるたびに，またプログラムからデバッガに制御が戻るたびに，プロセスIDを切り替えます。プロセスIDの切り替えは，DOSファンクションコールを使って行ないます。

DOSはリエントラントではないため，DOSの内部にブレークポイントを設定したり，DOSの内部をトレースすると，トラブルが発生することがあります。

DOSの内部に書き込みを行ないたいときは，-i-コマンドラインオプションを使ってプロセスIDの切り替えをオフにしなければなりません。ただしそうすると，プログラムとTurbo Debuggerがファイルハンドルを共用することになり，プログラムまたはTurbo Debuggerのファイルハンドルが足りなくなることがあります。

8087/80287数値演算コプロセッサとエミュレータ

Turbo Debuggerは，数値演算コプロセッサもソフトウェアエミュレータも使いません。両方ともプログラムが自由に使用できます。そのため浮動小数点数を扱うプログラムの動作は，デバッガのもとで動作しているときも変化しません。

割り込み

Turbo Debuggerは，デバッグしているプログラムの実行を制御するために，いくつかの割り込み処理に対して自分自身の割り込み処理を介挿(intercept)します。次の説明により，デバッグしているプログラムの割り込み処理に，Turbo Debuggerの介入があるかどうか判断できるでしょう。

INT 1，INT 3：

デバッガは，ブレークポイントや1命令ずつの実行を処理するために，これらの割り込みを使います。デバッグしているプログラムがこれらの割り込みを変更すると，その直後のブレークポイントでTurbo Debuggerが制御を取り戻せない可能性があります。普通のアプリケーションは，これらの割り込みをほとんど使いません。なぜならこれらの割り込みは，デバッガのように他のプログラムの実行を制御しなければならないプログラムのために用意されたものだからです。

INT 2：

多くのハードウェアデバッガは，この割り込みを使って，設定された条件に合致する事象が発生したことを知らせます。そのため，デバッグしているプログラムがこの割り込みを使用すると，この方式のボードやそれをサポートするためのデバイスドライバが，正しく動作しないことがあります。どうしてもこの割り込み使わなければならない場合には，プログラムの処理ルーチンを，その前の処理ルーチンの後にチェインしてください。

INT 9：

この割り込みは，キーの押下と解放を監視するために使用されるキーボードハードウェア割り込みです。デバッガは，ユーザプログラムの動作中は，プログラムの割り込み処理の前に自分自身の割り込み処理を介挿します。それにより，プログラムがループにはまり込んでいるときも，プログラムから制御を取り戻すことができます。この割り込みベクターを変更するプログラムが再起動されたときも正しく動作するように，デバッガは，プログラムが再起動されるたびに，このベクターを元の値に戻します。

INT 1とINT 3を使うプログラムのデバッグ

これらの割り込みを使用するプログラムをデバッグするときは，そのプログラムのデバッグ用バージョンでは，ロードしなければならないときにだけ割り込みベクターをロードし，不要になったらすぐに元の値に戻すべきです。この手法を用いると，デバッグできないコードの数を最小限まで減らすことができます。プログラムによりこれらのベクターが変更されている間は，デバッガを使って，プログラムコードを1ステップずつ実行したり，ブレークポイントにより実行を停止したりすることはできません。

ディスプレイモードの退避と切り替え

Turbo Debuggerは，プログラムを実行するときはいつでも，プログラムのディスプレイモードの退避と回復を行ないます。そのため，標準のROM BIOSコールを使ってディスプレイモードを変更している場合には問題は起きませんが，ディスプレイコントローラのレジスタを直接操作している場合は，デバッガにより設定が壊される可能性があります。

メモリ上の配置

デバッガを起動するとき，DOSは，自分自身と常駐プログラムの上位の最初の空きメモリにデバッガをロードします。次にデバッガが，自分自身のプログラムコードの上位に，作業用スタック領域とヒープ領域を確保します。そしてその上位に，デバッグするプログラムのシンボルテーブルをロードし，さらにその上位にプログラムをロードします。

デバッガを終了してDOSに戻るとき，デバッガは，デバッグしているプログラムとシンボルテーブルがロードされているメモリを解放します。デバッグしていたプログラムがDOSのメモリ領域確保ファンクション(48)を使ってメモリブロックを確保していたときは，そのメモリも解放します。

EMSのサポート

システムにEMS(Expanded Memory Specification)ボードが装備されているときには，Turbo Debuggerは，デバッグするプログラムのためにより多くのメインメモリを空けておくことができるように，プログラムのシンボルテーブルを拡張メモリに入れます。

またEMSメモリを使用するプログラムをデバッグすることができるように，EMSドライバの状態を退避したり回復したりします。

デバッグするプログラムがEMSメモリの全体を使用する場合，プログラムとデバッガがどちらもEMSを使用することにより互いに干渉し合うときは，インストールユーティリティTDINSTにより，デバッガがEMSを使用しないように設定することができます。

割り込みベクターの退避と回復

Turbo Debuggerは，下位メモリの最初の48個の割り込みベクター(00から2F)を3セット保存します。

1つ目のセットは，Turbo DebuggerがDOSのコマンドラインから起動したときに，これらのベクターからコピーされます。このセットは，***F10***/File/Quit(または***GRPH-X***)コマンドを選択してDOSに戻るとき，またはプログラムのデバッグ中に***F10***/File/OS shellコマンドを使ってDOSのコマンドを実行するときに，割り込みベクターを元の値に戻すために使用されます。

2つ目のセットはTurbo Debuggerのベクターです。Turbo Debuggerが動作しているときはいつでも，これらのベクターが使用されています。またプログラムの実行後にデバッガが制御を取り戻すとき，これらのベクターがセットされます。

3つ目のセットは，デバッグしているプログラムのベクターです。これらのベクターは，プログラムが実行（普通に，または1ステップずつ）されるときにセットされ，実行が停止してTurbo Debuggerが制御を取り戻すときに退避されます。この方法により，Turbo Debuggerが自分自身の割り込みベクターを使いながら，割り込みベクターを変更するプログラムをデバッグすることが可能になります。

付録D

組み込みアセンブラのキーワード

この付録には，組み込みアセンブラを使って命令を入力するときに使用する命令ニーモニックとその他の特別なシンボルが掲載されています。掲載されているキーワードは，Turbo AssemblerやMASMで使用されるものと同じです。

表D.1　8086/80186/80286命令ニーモニック

AAA	INC	LIDT **	REPNZ
AAD	INSB *	LLDT **	REPZ
AAM	INSW *	LMSW **	RET
AAS	INT	LOCK	REFT
ADC	INTO	LODSB	ROL
ADD	IRET	LODSW	ROR
AND	JB	LOOP	SAHF
ARPL **	JBE	LOOPNZ	SAR
BOUND *	JCXZ	LOOPZ	SBB
CALL	JE	LSL **	SCASB
CLC	JL	LTR **	SCASW
CLD	JLE	MOV	SGDT **
CLI	JMP	MOVSB	SHL
CLTS **	JNB	MOVSW	SHR
CMC	JNBE	MUL	SLDT **
CMP	JNE	NEG	SMSW **
CMPSB	JNLE	NOP	STC
CMPSW	JNO	NOT	STD
CWD	JNP	OR	STI
DAA	JO	OUT	STOSB
DAS	JP	OUTSB *	STOSW
DEC	JS	OUTSW *	STR **
DIV	LAHF	POP	SUB
ENTER *	LAR **	POPA *	TEST
ESC	LDS	POPF	WAIT
HLT	LEA	PUSH	VERR **
IDIV	LEAVE *	PUSHA *	VERW **
IMUL	LES	PUSHF	XCHG
IN	LGDT **	RCL	XLAT
			XOR

*　CPUが186および286プロセッサのときのみ使用可能

**　CPUが286プロセッサのときのみ使用可能

表D.2　8087/80287数値演算コプロセッサ命令ニーモニック

FABS	FIADD	FLDL2E	FST
FADD	FICOM	FLDL2T	FSTCW
FADDP	FICOMP	FLDPI	FSTENV
FBLD	FIDIV	FLDZ	FSTP
FBSTP	FIDIVR	FLD1	FSTSW *
FCHS	FILD	FMUL	FSUB
FCLEX	FIMUL	FMULP	FSUBP
FCOM	FINCSTP	FNOP	FSUBR
FCOMP	FINIT	FNSTS **	FSUBRP
FCOMPP	FIST	FPATAN	FTST
FDECSTP	FISTP	FPREM	FWAIT
FDISI	FISUB	FPTAN	FXAM
FDIV	FISUBR	FRNDINT	FXCH
FDIVP	FLD	FRSTOR	FXTRACT
FDIVR	FLDCW	FSAVE	FYL2X
FDIVRP	FLDENV	FSCALE	FYL2XP1
FENI	FLDLG2	FSETPM *	F2XM1
FFREE	FLDLN2	FSQRT	

*　80287コプロセッサに対してだけ使用可能

**　80287コプロセッサのときは，fstsw命令およびfnstsw命令は，通常のメモリオペランドだけでなく，AXレジスタもオペランドに指定することができます。

表D.3　CPUのレジスタ

バイトレジスタ	ah, al, bh, bl, ch, cl, dh, dl
ワードレジスタ	ax, bx, cx, dx, si, di, sp, bp
セグメントレジスタ	cs, ds, es, ss
浮動小数点レジスタ	st, st(0), st(1), st(2), st(3), st(4), st(5), st(6), st(7)

表D.4　特別なキーワード

WORD PTR	**TBYTE PTR**
BYTE PTR	**NEAR**
DWORD PTR	**FAR**
QWORD PTR	**SHORT**

付録E

Turbo Debuggerのカスタマイズ

Turbo Debuggerは，マスタディスクから作業用コピーを作成するとそのまま動作するように設定されていますが，インストールプログラムTDINST.EXEを使うと，多数のオプションの設定を変更することができます。またオプションのいくつかは，Turbo Debuggerを起動するときに，コマンドラインオプションを使って変更することができます。オプションの一時的な変更にはコマンドラインオプションを使う方が便利ですが，同じコマンドラインオプションを繰り返し指定するときは，インストールプログラムを使って設定しておく方が効率的です。

TDINSTプログラムで設定できる項目は次のとおりです。

■ウィンドウの色

■ディスプレイパラメータ：起動時の表示，画面の切り替え方式，整数の表示書式，ログリストの長さ，タブ間隔，Watchウィンドウの最大サイズ，スノー制御，46行(43/50行)モード，グラフィックス画面の保存モード

■ユーザのエディタを起動するためのコマンド

■ソースファイルを探すディレクトリ

■キーボードパラメータ：プログラムの割り込みキー，***CTRL***キーショートカット

■プロンプトパラメータ：エラー発生時のビープ音，***ESC***によるエラーメッセージの消去，ヒストリリストの長さ

■大文字と小文字の区別

■DOSのプロセスIDの切り替え

■EMSメモリへのシンボルテーブルの格納

TDINSTの起動とオプションの設定方法

DOSプロンプトでTDINSTと入力して**リターン**を押すと，インストールプログラムを起動することができます。メインメニューのオプションを選択するには，各オプションのハイライト表示された先頭文字を押すか，上下の矢印キーを使って選択したいオプションにハイライトバーを移動して**リターン**を押します。たとえば，表示の設定を変更するには***D***を押します。他のメニューのオプションも同じ方法で選択することができます。***ESC***を押すと1つ前のメニューに戻ります。メインメニューに戻るために複数回押さなければならない場合もあります。

画面の表示色の設定

メインメニューの中で***C***を押すと，色選択メニューが表示されます。このメニューでは，Customize, Default color set, Two tone color setの3つのオプションを選択できます(IBM PCでは，Two tone color setのかわりにAlternate color set，Turquoise color set，Black color setの3つがあり，合計5つのオプションになります)。それぞれ異なる色を指定することができる非常にたくさんの画面項目があるので，2番目以降のオプションを指定して，デフォルトのカラーセットの1つを選択したい人が多いでしょう。デフォルトのカラーセットがどれも気に入らないときは，***C***を押してCustomizeを選択することにより，全面的に自分自身の好みの色に設定できます。

またデフォルトのカラーセットを選択して，その一部を変更することもできます。それには，***D***または***T***（IBM PCでは***D***，***A***，***T***，***B***）を押してデフォルトのカラーセットの1つを選択してから，Customizeを選択します。そうすると，いま選択したカラーセットの設定を変更することができます。

Cutomizeを選択すると，新しいメニューが現われて次のオプションが表示されます。

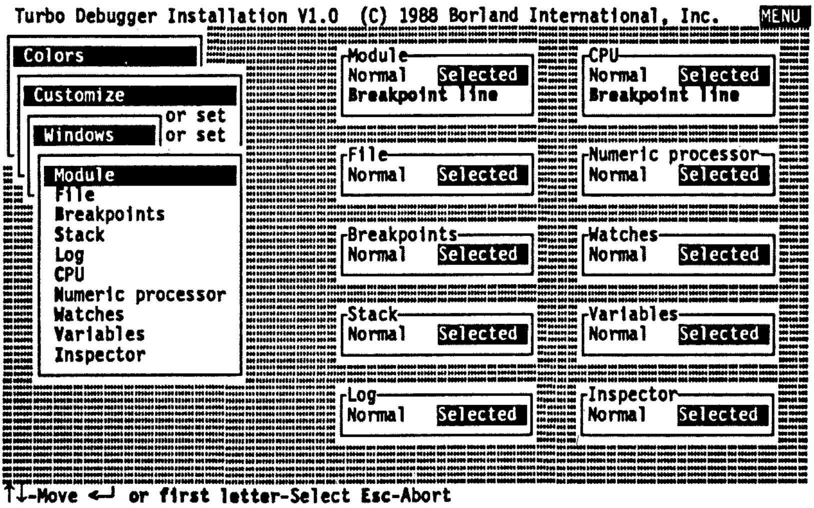
Windows　　Turbo Debuggerがプログラムに関する情報を表示する各種のウィンドウの色を選択することができます。

User input　　入力を求める各種のボックス（メニュー，エラーメッセージ，ヘルプ，ヒストリリストなど）の色を選択することができます。

Screen　画面のバックグラウンド(背景)のパターン(模様)と色を選択することができます。またヘルプ行（クイックリファレンス行）の色を選択することもできます。

これらの項目を選択するとさらにもう1つのメニューが表示され,色を変更したいウィンドウやボックスを選択することができます。変更したい項目を選択すると，メニューを覆うようにパレットボックスが表示されます。カラーディスプレイを使用しているときは,カラーパレットの形式で，7つの色とそれらの反転が表示されます(IBM PCでは，16の前景色と8色の背景色のすべてが表示されます。白黒のディスプレイを使っているときは，下線，反転表示，ハイライト表示などのいろいろな属性の組み合わせを選択することができます)。

図E.1　ウィンドウの色のカスタマイズ



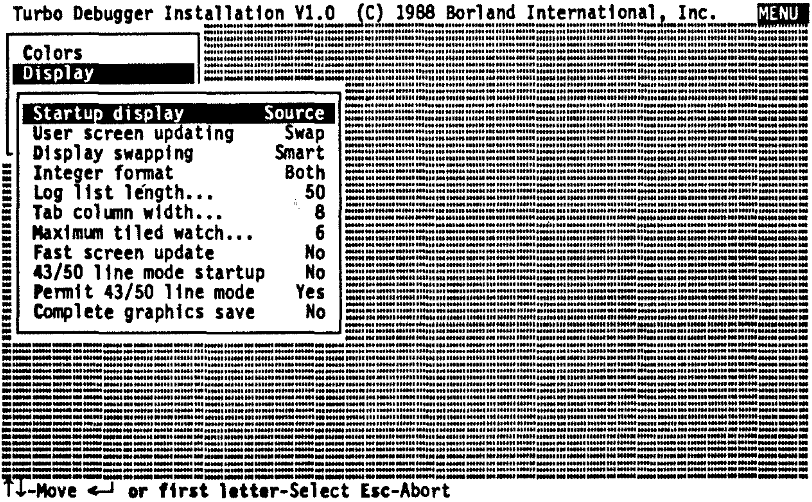
矢印キーを使うと，パレットボックスの中を動き回ることができます。メニューおよびパレットの表示領域の右側には，ウィンドウの色を変更しているかボックスの色を変更しているかにより，すべての種類のウィンドウまたはボックスが表示されます。矢印キーを使ってパレットボックスの中を動き回ると，それにつれて，色を変更しているウィンドウの色が更新され，現在選択している色でどのように見えるかが示されます。気に入ったら**リターン**を押すと，その色に設定することができます。

注意：［**IBM PC**］Turbo Debuggerは，カラー表示，白黒表示，モノクローム表示のために1つずつ3つのカラーテーブルを保存します。カラーテーブルは，現在のビデオモードとディスプレイハードウェアに基づいて選択された1つのカラーテーブルしか変更できません。したがって，カラーディスプレイを使用していて白黒のカラーテーブルを変更したいときは，まずDOSのプロンプトでMODE BW80と入力してビデオモードを白黒モードに変更してから，TDINSTを起動してください。なお，PC-9801では，TD.EXEはただ1つのカラーテーブルしか保存していません。

ディスプレイパラメータの設定

メインメニューから**D**を押すとDisplayメニューが現われます。このメニューのオプションのいくつかは，Turbo Debuggerを起動するときにDOSのコマンドラインから設定することもできます。またTDINSTを使わないと設定できないものもあります。

図E.2　ディスプレイパラメータのカスタマイズ



Beginning displayオプション

Turbo Debuggerが起動したときの画面表示を変更することができます。このオプションを選択するたびに，次の2つの設定に交互に切り替わります。

Assembler　　**アセンブラ起動**：CPUウィンドウにプログラムの逆アセンブルした命令が表示されます。どの命令も実行されません。

Source　　**ソース起動**：Moduleウィンドウにプログラムのソースコードが表示されます。コンパイラスタートアップコードが実行され，プログラムの先頭

のソースコードから実行することができます。

User screen updatingオプション

Turbo Debuggerの画面とプログラムの画面の切り替え方式を選択することができます。このオプションを選択するたびに，次の2つ（IBM PCでは3つ）の方式に順番に切り替わります。

Flip pages

Turbo Debuggerの画面を，テキストVRAMの2画面目（もう一方のディスプレイページ）に置きます。これによって，ユーザプログラム画面（1画面目）との切り替えが高速に行なわれます。

PC-9801：ほとんどの日本語入力フロントプロセッサは，変換モードの起動時に強制的にアクティブなバンクを＃0（1画面目）に切り替えるため，Turbo Debuggerの中でそうしたフロントプロセッサを使う場合には，日本語入力モードに入ったときにユーザプログラム画面に切り替わってしまいます。

IBM PC：このオプションは，使用しているディスプレイアダプタがCGA, EGA, VGAのような複数のディスプレイページを備えている場合にのみ有効です。モノクロームディスプレイを使用しているときは，このオプションは使えません。このオプションは，ほとんどのデバッグで問題なく動作します。プログラムが複数のディスプレイページを使う場合には，プログラムの表示が乱れる可能性がありますが，そのようなプログラムは少数です。

Swap

単一のディスプレイアダプタとディスプレイページを使って，退避と再ロードによりTurbo Debuggerの画面とプログラムの画面を切り替えます。このオプションは，表示の切り替えは最も低速ですが，確実に動作します。複数のディスプレイページを使用するプログラムをデバッグするときは，このオプションを使ってください。

またTurbo Debuggerの内部から他のユーティリティを起動する場合や，SideKickのようなTSRをロードしている場合にも，このオプションを使ってください。

Other display（IBM PCのみ）

システムのもう1つのディスプレイにTurbo Debuggerの画面を表示します。カラーディスプレイアダプタとモノクロームディスプレイアダプタの両方を使用しているとき

は，このオプションを使うと，一方のディスプレイにプログラムの画面を，他方のディスプレイにTurbo Debuggerの画面を表示させることができます。

Display swappingオプション

Turbo Debuggerの画面からプログラムの画面への切り替えモードを選択することができます。このオプションを選択するたびに，次の3つのモードに順番に切り替わります。

None

画面を切り替えません。ディスプレイに何も出力しないプログラムをデバッグするときは，このオプションを使ってください。

Smart

表示出力が発生する可能性があるときだけ，ユーザ画面に切り替えます。Turbo Debuggerは，ビデオメモリに対して読み出しや書き込みを行なうように見えるルーチンがスキップされたとき，またはそのような命令やソース行が実行されたときに画面を切り替えます。このオプションがデフォルトです。

Always

ユーザプログラムが実行されるたびに，ユーザ画面に切り替えます。Smartオプションではプログラムの画面への書き込みを完全に捉えることができないときに，このオプションを使ってください。このオプションを選択したときは，プログラムを1命令または1行実行するたびに，画面がちらつきます。Turbo Debuggerの画面が，短時間にプログラムの画面に切り替わるからです。

Integer formatオプション

整数の表示書式を変更することができます。このオプションを選択するたびに，次の3つの書式に順番に切り替わります。

Hex	16進数で表示します。
Decimal	10進数で表示します。
Both	16進数と10進数の両方で表示します。

Log list lengthオプション

各プロンプトのヒストリリストに保存される応答の数を設定することができます。このオプションを選択すると，保存する応答の数を入力するように求めます(最大は200，最小は4です)。

Tab column widthオプション

テキストファイルまたはソースファイルが表示されているときの，タブストップ間の桁数(間隔)を設定することができます。このオプションを選択すると，桁数の入力を求めます。1から32までの数を入力できます。

Maximum tiled watchオプション

並置表示モードにおいてWatchウィンドウが拡大可能な最大行数を設定することができます。このオプションを選択すると，行数の入力を求めます。1から20までの数を入力できます。

Screens linesオプション

起動時の画面の表示行数を設定することができます。46行モードで起動したいときはこのオプションの設定をYesにしてください。IBM PCでは，EGAまたはVGAを使用しているときには，43行または50行の表示モードを選択することができます。

Fast screen updateオプション（IBM PCのみ）

カラーディスプレイの更新速度を変更することができます。スノーノイズ（画面のちらつき）が気になるときは，このオプションをNoにしてください（スノーノイズが多くても，更新速度が速い方を好む人もいます)。

Permit 43/50 line modeオプション（IBM PCのみ）

大型ディスプレイモードを可(Yes)または不可(No)に交互に切り替わります。このオプションをNoに設定すると，約8Kのメモリを節約できます。なぜなら，大型ディスプレイモードを使用するときは，デバッガの中により大きなウィンドウバッファスペースが必要だからです。このオプションは，実行するためにできるかぎり多くのメモリを必要とする，非常に大きなプログラムをデバッグするときに便利です。このオプションがNoに設定されているときは，システムが43/50行の表示ができる場合にも，43/50行モードに切り替えることはできません。

PC-9801でこれに対応するメニュー項目は，Mode for hardware/Screen modeオプションになります。

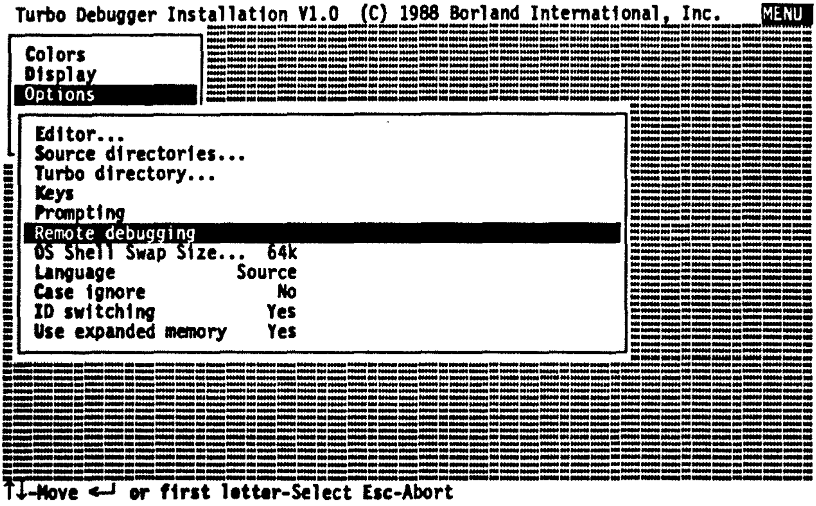
Complete graphics saveオプション（IBM PCのみ）

プログラムの画面とTurbo Debuggerの画面を切り替えるときに，グラフィックディスプレイバッファの内容をそのまま退避するかしないかを切り替えることができます。このオプションをNoにすると，約12Kのメモリを節約することができます。このオプションは，実行するためにできるかぎり多くのメモリを必要とする，非常に大きなプログラムをデバッグするときに便利です。このオプションをNoにしても，プログラムの画面がわずかに損われるだけで，デバッグに支障はありません。

Turbo Debuggerのオプションの設定

TDINSTのメインメニューで**O**を押すと，Optionsメニューが表示されます(このメニューのオプションのいくつかは，Turbo Debuggerのコマンドラインからも設定できます)。

図E.3　Optionsメニューのオプションをカスタマイズする



Editorオプション

ユーザのエディタを起動するDOSコマンド（エディタの.EXEまたは.COMファイル名）を設定することができます。このオプションにより，デバッグ中にファイルを編集したいときに，自分の好みのエディタを起動させることができます。

Turbo Debuggerは，このコマンドの後に1つの空白を置いて，編集するファイルの名前を付け加えます。

Source directoriesオプション

Turbo Debuggerがソースファイルを探すディレクトリのリストを設定することができます。

Turbo directoryオプション

Turbo Debuggerがヘルプファイルおよびコンフィギュレーションファイルを探すディレクトリを設定することができます。

Keysオプション

プログラムの実行を中止させるための割り込みキーをどのキーにするか設定することができます。また***CTRL***キーをどの目的に使用するか選択することができます。

Key for interrupt

プログラムの実行中に割り込みをかけるキーを，***CTRL***キーとどのキーの組み合わせにするかを指定します。STOPおよびESCのどちらかを選択します（IBM PCではBreak，Esc，NumLock，SysReqのどれか）。
Other keyを選択すると，使用したいキーの組み合わせを直接押すことにより，そのキーの組み合わせを割り込みキーとして設定できます。***SHIFT***，***GRPH***（***Alt***），***CTRL***と英字，ファンクションキー，その他の普通キーの任意の組み合わせを使用できます（IBM PCでは，**左*Shift***と**右*Shift***は区別されます）。

注意：IBMの拡張キーボード(101キー)に対しては，**左*Alt***キーと**右*Alt***キー，**左*Ctrl***キーと**右*Ctrl***キーを区別しません。ただし**左*Shift***キーと**右*Shift***キーは，異なるキーとして扱います。

Control-Key shortcut

CTRLキーショートカットの使用を可（Yes）または不可（No）に交互に切り替えます。***CTRL***キーショートカットを使用可能にしたときは，***CTRL***キーとローカルメニュ

ーコマンドの先頭文字を一緒に押すことにより，そのコマンドを直接発行することができます。ただしこの場合には，それらのキーの組み合わせは，WordStar方式のカーソル移動コマンドとして使用することができません。

Promptingオプション

エラー発生時のビープ音，***ESC***によるエラーメッセージの消去，ヒストリリストの長さなどを設定することができます。設定を変更しようとするメニュー項目の先頭文字を押してください。

Beep on error

ビープ音を鳴らす（Yes）または鳴らさない（No）に交互に切り替えます。ビープを鳴らすように設定したときは，エラーメッセージが表示されたとき，および間違ったキーを押したときにブザーが鳴ります。

Error message clearing

ESCを選択すると，エラーメッセージは***ESC***を押さなければ消えません。Nextを選択すると，何かキーを押して新しいコマンドを発行したときに自動的に消えます。

History list

各プロンプトのヒストリリストに保存する入力の数を設定することができます。

Remote debuggingオプション

リモートリンクを可または不可に切り替えることができます。またリモートシステムと通信するためのシリアルポートや通信速度を選択することができます。

Remote debugging

リモートリンクを可(Yes)または不可(No)に交互に切り替えます。通常このオプションはNoに設定したいでしょう。なぜならYesに設定すると，Turbo Debuggerは，起動したときにいつでもリモートリンクを確立しようとするからです。

Port

リモートリンクのために使用するシリアルポートをCOM1, COM2, またはCOM3に交互に切り替えます（IBM PCではCOM1とCOM2のみ）。

Speed

リモートリンクの通信速度を次の2つ（IBM PCでは3つ）の間で切り替えます。

PC-9801：	9600bps	40/31Kbps	
IBM-PC：	9600bps	40Kbps	115Kbps

OS shell swap sizeオプション

Turbo Debuggerの内部からDOSのコマンドを入力するときに，Turbo Debuggerがスワップにより用意する空き領域の大きさを設定することができます。

Languageオプション

Turbo Debuggerが式を評価するために使用する言語を指定することができます。選択できる言語は次のとおりです。

Source module

カレントソースモジュールの言語を使用します。

C

カレントソースモジュールの言語によらずに，いつでもCを使用します。

Pascal

カレントソースモジュールの言語によらずに，いつでもPascalを使用します。

Assembler

カレントソースモジュールの言語によらずに，いつでもアセンブラを使用します。

Ignore caseオプション

大文字と小文字の区別を無視する（Yes），または無視しない（No）に交互に切り替わります。

Change process IDオプション

プロセスIDの切り替えを行なう（Yes），または行なわない（No）に交互に切り替わります。付録Cに説明されている技術的な問題を良く理解していてDOSの内部までトレースするとき以外は，このオプションをNoに設定しないでください。

Use expanded memoryオプション（IBM PCのみ）

シンボルテーブルを格納するためにEMSメモリを使用する（Yes），または使用しない（No）に交互に切り替わります。デバッグするプログラムがEMSを使う場合にも，このオプションをYesに設定することができます。

Mode for hardwareメニュー（PC-9801）

このメニューでは，Turbo Debuggerの画面にハードウェアのどの画面を使用するか，シンボルテーブルをどこに置くか，ユーザプログラムのグラフィックス画面をどのように扱うかを設定します。

Screen modeオプション

Turbo Debuggerの画面表示に，テキスト画面のみを使うか，グラフィック画面も使用するか，グラフィック画面も使用する場合にはGVRAMのどちらのバンクを使うのかを選択します。

Text only

Text onlyを選択すると，グラフィック画面による46行モードは使用できなくなります。GVRAMを2バンクとも使用するプログラムをデバッグする場合は，Text onlyに設定しておく必要があります。また，シンボルテーブルをGVRAMのバンク＃1に置いて，グラフィックスを用いるプログラムをデバッグする場合にも，Text onlyにしておかなければなりません。

Foreground (bank ＃0) graphics and text

46行モードの画面用にGVRAMのバンク＃0を使用します。46行モードを使用したいときに，次のような制約または要請がある場合にはこのオプションを選択してください。

■最初期型のPC-9801やPC-9801Uなど，GVRAMのバンクが1枚しかないシステムを使用している。

■GVRAMのバンク＃1にはシンボルテーブルを置きたい。

■GVRAMのバンク＃1はRAMディスクとして使用したい。

Background (bank ＃1) graphics and text

46行モードの画面用にGVRAMのバンク＃1を使用します。通常の（GVRAM ＃0しか使わない）グラフィックスプログラムのデバッグには，このオプションが適しているでしょう。

Buffer of symbolオプション

シンボルテーブルを格納しておくメモリを，メインメモリ(**M**emory)，GVRAMのバンク#1(**B**ackground)，あるいはEMSの中から選択します。デバッグするプログラムがEMSを使用する場合でも，EMSを選択することができます。

Graphics switchesオプション

ユーザプログラムのグラフィック画面を表示(**E**nable)するようにするか，非表示(**D**isable)とするかを指定します。Enableにした場合，TDのOptions/**H**ardware statusメニューの**V**isual user G-pageでセットされているバンクの内容が，つねに（デバッガの画面においても）表示されることになります。Disableとした場合でも，デバッガのグラフィック画面(46行画面）が表示されなくなることはありません。

Mode for displayメニュー（IBM PC）

デバッガが使用するビデオモードを次の中から選択します。

Default

TD.EXEの起動時にシステムに接続されているビデオアダプタを調べて，カラーアダプタが検出された場合には，TD.EXEがロードされたときにアクティブなビデオモードで動作を行ないます。

Color

起動時にカラーアダプタが検出されれば，どのモードがアクティブかにかかわらず，80桁カラーモードで動作し，終了するときには起動時にアクティブだったモードに戻します。

Black and white

起動時にカラーアダプタが検出されれば，どのモードがアクティブかにかかわらず，80桁白黒モードで動作し，終了するときには起動時にアクティブだったモードに戻します。

Monochrome

モノクロームアダプタが検出されれば，どのモードがアクティブかにかかわらず，モノクロモードで動作し，終了するときには起動時にアクティブだったモードに戻します。

LCD

起動時にカラーアダプタが検出されれば，どのモードがアクティブかにかかわらず，80桁白黒モードで動作し，終了するときには起動時にアクティブだったモードに戻します。

コマンドラインオプションとTDINSTのメニューオプションの対応

ここまでの節で説明したオプションのいくつかは，Turbo Debuggerを起動するときにコマンドラインオプションによりオーバーライドすることができます。Turbo Debuggerのコマンドラインオプションと，そのオプションを半永久的に設定するインストールプログラムのコマンドの対応関係を次表に示します。

コマンドラインオプション	TDINSTのメニューの選択
-do	Display/User screen updating/Other display (IBM PC)
-dp	Display/User screen updating/Flip pages
-ds	Display/User screen updating/Swap
-g, **-g**＋	Mode for hardware/Graphics switch/Enable (PC-9801)
-g-	Mode for hardware/Graphics switch/Disable (PC-9801)
-g0	Mode for hardware/Screen mode/Foreground (PC-9801)
-g1	Mode for hardware/Screen mode/Background (PC-9801)
-i	Options/Change process ID/Yes
-i-	Options/Change process ID/No
-l	Display/Beginning display/Assembler
-l-	Display/Beginning display/Source
-r	Options/Remote debugging/Remote debugging/Yes
-r-	Options/Remote debugging/Remote debugging/No
-rp＃	Options/Remote debugging/Port/＃
-rs＃	Options/Remote debugging/Speed/＃
-sd	Options/Source directories
-sc	Options/Ignore case/Yes
-sc-	Options/Ignore case/No
-vn	Display/Permit 43/50 line mode/Yes (IBM PC)
-vn-	Display/Permit 43/50 line mode/No (IBM PC)
-vg	Display/Complete graphics save/Yes (IBM PC)
-vg-	Display/Complete graphics save/No (IBM PC)

設定の保存とTDINSTの終了

設定の変更が終了したら，メインメニューに戻るのに必要な回数だけ***ESC***を押しメインメニューに戻ってください。

メインメニューには，SaveコマンドとQuitコマンドがあります。Saveコマンドを選択するとメニューが現われて，それまでに行なった設定をTurbo Debuggerの実行可能なプログラムファイルTD.EXEに直接書き込むか，コンフィギュレーションファイルに保存するかを選択することができます。

コンフィギュレーションファイルへの保存を選択すると，プロンプトが現われて，デフォルトのコンフィギュレーションファイルTDCONFIG.TDが表示されます。**リターン**を押してこのファイル名を受け入れることもできますし，新しいコンフィギュレーションファイル名を入力することもできます。新しいファイル名を入力したときは，Turbo Debuggerを起動するときに，-**c**コマンドラインオプションを使って，そのコンフィギュレーションファイルをロードすることができます。たとえば次のように指定します。

```
td -cmycfg myprog
```

またTurbo Debuggerを起動した後は，Options/Restore optionsコマンドを使って，コンフィギュレーションファイルをロードすることができます。

出荷時の標準の設定に戻したいときは，TD.EXEをマスタディスクから作業用ディスクにコピーしてください。

付録F

ハードウェアデバッガインターフェース

ハードウェアデバッグボードを使用すると，いくつかの種類のブレークポイント，特にメモリのある領域の内容やプログラム変数の値の変化を監視するブレークポイントをセットしたときの実行速度が大幅に向上します。

この付録では，Turbo Debuggerと一緒に提供されるデバイスドライバのインストールの仕方と，ハードウェアデバッガの機能を利用するためにTurbo Debuggerがハードウェアデバッガと通信できるようにするためのデバイスドライバの書き方を説明します。

注意：現在のところ，PC-9801用のTurbo Debuggerがサポートするハードウェアデバッグボードはありません。

80386ハードウェアデバイスドライバ

この付録の情報は，Turbo Debuggerが自社のハードウェアデバッガを利用できるようにしたいと望まれるベンダの方々を対象としています（DOSのデバイスドライバの一般的なアーキテクチャについての知識が必要です。デバイスドライバの書き方の詳しい説明は，DOSの技術解説書を読んでください）。

Turbo DebuggerのマスタディスクにはTDH386.SYSが入っています。このファイルは，Turbo Debuggerが80386プロセッサのデバッグレジスタを利用できるようにするためのハードウェアデバイスドライバです。このデバイスドライバを使うときは，マスタディスクからDOSのディスクにコピーし，CONFIG.SYSファイルに次の行を追加してください。

```
DEVICE = TDH386.SYS
```

このように設定すると，Turbo Debuggerは，ブレークポイントの処理をスピードアップするために，可能なときはいつでも80386のハードウェアの助けを利用します。このことからも明らかなように，このデバイスドライバは，CPUが80386プロセッサのシステムでしか使用できません。

ハードウェアブレークポイントの設定方法

ハードウェアに支援されるブレークポイントは，次の3つの方法でセットすることができます。

- ■**F10**/Breakpoints/Changed memory globalコマンドを使う。
- ■BreakpointsウィンドウのローカルメニューのCondition/Changed memoryコマンドを使う。
- ■BreakpointsウィンドウのローカルメニューのCondition/Hardwareコマンドを使う。

Changed memoryコマンドのひとつを使ってブレークポイントをセットしたときは，Turbo Debuggerは自動的に，そのブレークポイントのためにハードウェアデバッガを利用できるかどうかを調べます。利用できるときはハードウェアブレークポイントをセットします。そしてBreakpointsウィンドウの左側のペインのグローバルブレークポイント番号の後にアスタリスク(*)を表示して，そのブレークポイントがハードウェアの中にセットされたことを示します。

次の2つの節では，一般化されたハードウェアブレークポイントの設定方法について説明します。まずTDH386.SYSデバイスドライバを使ってセットできるブレークポイントの種類を説明します。次に他のハードウェアデバイスドライバを使ってセットできるハードウェアブレークポイントオプションについて説明します。お手持ちのデバイスドライバのより詳細な情報は，そのデバイスドライバのベンダのマニュアルを参照してください。

TDH386.SYSにより設定できるハードウェアブレークポイントのマッチング条件

TDH386.SYSをTurbo Debuggerの普通バージョンと一緒に使うときは，BreakpointsウィンドウのローカルメニューのCondition/Hardwareコマンドから，次のハードウェアブレークポイントをセットすることができます。

■Fetch instruction（命令フェッチにマッチ）
■Read memory（メモリからの読み込みにマッチ）
■Access memory（メモリからの読み込みおよびメモリへの書き込みにマッチ）

TDH386.SYSを使っているときは，データマッチオプションはいつでもMatch allにセットしなければなりません。またアドレスマッチオプションは，単一のメモリアドレス(equal)かアドレス範囲(Range)にしかセットできません。アドレス範囲の大きさは1から16バイトで，すでにセットされているブレークポイントの数と先頭アドレスによって変化します。

BreakpointsウィンドウのHardwareメニューの他のオプションは，もっと複雑なマッチングモードをサポートするハードウェアデバッガとデバイスドライバのためのものです。

TDH386.SYSは，仮想デバッガ(TD386)とともに使用することもできます。この場合には，任意の大きさのアドレス範囲やI/Oアクセスにマッチするブレークポイントをはじめとする，多種類のハードウェアブレークポイントを使うことができます。仮想デバッグについての詳しい説明は第14章を参照してください。

BreakpointsウィンドウのHardwareメニュー

この節ではBreakpointsウィンドウのローカルメニューのConditionオプションから選択できるHardwareメニューについて説明します。使用されているハードウェアは，おそらくこのメニューから指定できるマッチングのすべての組み合わせをサポートすることはできないでしょう。前の節では，Turbo Debuggerと一緒に提供されるTDH386.SYSデバイスドライバを使うときに指定できる組み合わせについて説明しました。

Hardwareメニューからは，ハードウェアブレークポイントを構成する3つの条件をセットすることができます。

- ■マッチするバスサイクルの種類
- ■マッチするアドレスの範囲
- ■マッチするデータの値の範囲

ハードウェアブレークポイントは，たとえば“3F8から3FFの範囲内のアドレスへのデータの値が1のI/O書き込みを監視する”ということになります。このブレークポイントは，COM1シリアルポートを制御するI/Oポートのどこかに，値が1のバイトが書き込まれたときはいつでもトリガされます。

通常セットするブレークポイントは，たとえば“アドレス200へのI/Oを監視する”というような，これよりもずっと簡単なブレークポイントが多いでしょう。

HardwareメニューのCycle typeオプションには，次のオプションがあります。

Read memory	メモリからの読み込みにマッチする。
Write memory	メモリへの書き込みにマッチする。
Access memory	メモリに対する読み書きにマッチする。
Input I/O	I/Oポートからの読み込みにマッチする。
Output I/O	I/Oポートへの書き込みにマッチする。
Both I/O	I/Oポートに対する読み書きにマッチする。
Fetch instruction	命令フェッチにマッチする。

Access memoryオプションは，Read memoryオプションとWrite memoryオプションの組み合わせで，メモリからの読み込みとメモリへの書き込みのどちらにもマッチします。同様にBoth I/Oオプションは，I/Oポートからの読み込みとI/Oポートへの書き込みの両方

にマッチします。

ハードウェアデバッガの中には，メモリからのデータの読み込みと命令のフェッチを区別できるものもあります。このようなハードウェアでは，メモリからのデータの読み込みにマッチするようにセットされたブレークポイントは，そのロケーションからの命令のフェッチによってはトリガされません。

命令サイクルには，オペコードだけでなく，プリフィックス，オペランドのアドレス，即値データなども含まれます。ただしオペランドアドレスにより指定されるメモリロケーションに対するデータの読み書きは，命令フェッチの一部とはみなされません。たとえば

```
mov    ax,[1234]
```

の場合には，命令サイクルで3バイトフェッチされ，その後データが2バイト読み込まれます。命令フェッチをマッチング条件として指定する場合は，80x86ファミリのプロセッサは実行すべき命令をプリフェッチすること，そのためハードウェアデバッガにプリフェッチされた命令と実際に実行された命令を識別する機能がないと，間違ったマッチングが発生する可能性があることに注意してください（パイプライン機構による）。

HardwareメニューのAddressオプションには，次のオプションがあります。

Above	指定されたアドレスより大きいときマッチ
Below	指定されたアドレスより小さいときマッチ
Range	指定されたアドレス領域内のときマッチ
Not range	指定されたアドレス領域外のときマッチ
Less or equal	指定されたアドレス以下のときマッチ
Greater or equal	指定されたアドレス以上のときマッチ
Equal	指定されたアドレスに等しいときマッチ
Unequal	指定されたアドレス以外のときマッチ
Match all	すべてのアドレスにマッチ

HardwareメニューのDataオプションには，次のオプションがあります。

Above	指定された値より大きいときマッチ
Below	指定された値より小さいときマッチ
Range	指定された値の範囲内のときマッチ
Not range	指定された値の範囲外のときマッチ
Less or equal	指定された値以下のときマッチ
Greater or equal	指定された値以上のときマッチ

Equal	指定された値に等しいときマッチ
Unequal	指定された値以外のときマッチ
Match all	すべての値にマッチ

ハードウェアデバッガの概要

デバイスドライバインターフェースを利用すると，ハードウェアに依存することなく共通の方法で，異機種のハードウェアデバッガの機能にアクセスすることが可能になります。この目的を達成するために，何種類かのハードウェアデバッガに共通の機能を組み合わせた，一般的なハードウェアデバッガを考えます。Turbo Debuggerからデバイスドライバに何かを要求するときは，この抽象化したモデルを使用します。

ハードウェアボードの中には，この抽象化されたインターフェースにより決められているすべての機能をサポートしないものもあります。その場合には，デバイスドライバは，Turbo Debuggerに要求が実行できないことを知らせることができます。またこのインターフェースにより決められているよりも多くの機能をサポートするものもあります。その場合には，決められていない機能は利用できません。

このデバイスドライバインターフェースには，将来新しい機能を取り入れることができるように，デバイスドライバがTurbo Debuggerからの要求に応じて返すステータスフィールドが決めてあります。このステータスフィールドを利用することにより，Turbo Debuggerはハードウェアボードがサポートする機能を知ることができます。また新しいデバイスドライバは，このインターフェースに将来追加される機能を利用することができます。さらに，新しいインターフェースと古いデバイスドライバとの互換性を維持することもできます。

このハードウェアデバッガインターフェースは，デバッガボードの機能を次の3つに分類しています。

1. メモリおよびI/Oへのアクセスにマッチするブレークポイント
2. 命令トレースバックメモリ
3. シンボルテーブルを格納するためのオンボードメモリ

抽象インターフェースの現バージョンは，第1の機能だけをサポートします。将来のバージョンでは，その他の機能にアクセスするためのインターフェースを決める予定です。デバイスドライバを書くときは，将来2番目と3番目の分野の機能がサポートされる可能性があることを念頭にいれておいてください。

デバイスドライバインターフェース

デバイスドライバは，普通のキャラクタタイプのデバイスドライバです。システムをブートするときにデバイスドライバがロードされるようにするには，CONFIG.SYSファイルに次の行を追加しなければなりません。

```
DEVICE=drvrname.ext
```

ここで*drvrname.ext*は，あなたが作成したデバイスドライバファイルの名前です。デバイスドライバは，次のファンクションコールをサポートしなければなりません。

INIT　コマンドコード＝0

デバイスドライバがロードされたときに，Turbo Debuggerにより一度だけ呼び出されます。このファンクションは，ハードウェアボードが使用不可（disable）にされ，動作停止状態にあることを確認しなければなりません。

READ　コマンドコード＝4

Turbo Debuggerからデバイスドライバに送られた最後のコマンドの結果を示すステータスブロックを読むために，Turbo Debuggerにより呼び出されます。したがってデバイスドライバの内部のデータエリアに最新のステータスを保存し，要求された数のバイトを返すようにしなければなりません。READが呼び出されるたびに，その前のREADに対してステータスブロックの全体を返さなかった場合にも，ステータスブロックの先頭から返さなければなりません。

次の節では，Turbo Debuggerからのいろいろなコマンドブロックに応答してデバイスドライバが返すことができる，いろいろなステータスブロックについて説明しています。

READNOWAIT　コマンドコード＝5

ステータスブロックの最初のバイトを返します。データが用意できていることを示すビジービットが0にセットされていなければなりません。

READSTATUS　コマンドコード = 6

いつでもビジービットを0にセットします。この後のREADコマンドは，ただちに実行されます。

READFLUSH　コマンドコード = 7

リターンステータスの終了ビットをセットします。

WRITE　コマンドコード = 8

デバイスドライバにコマンドを送るために，Turbo Debuggerにより呼び出されます。コマンドは，種類により長さが異なります。デバイスドライバの内部の作業領域にデータをコピーすることができますし，デバイスドライバの要求の一部であるデータポインタを使って直接にアクセスすることもできます。

次の節で，Turbo Debuggerがデバイスドライバに送ることができるいろいろなコマンドブロックについて説明しています。

WRITEVERIFY　コマンドコード = 9

WRITE(コマンドコード8)と同じことをします。

WRITESTATUS　コマンドコード = 10

リターンステータスの終了ビットをセットします。

WRITEFLUSH　コマンドコード = 11

リターンステータスの終了ビットをセットします。

他のすべてのファンクションコールに対しては，リターンステータスワードのエラービット(ビット15)をセットし，かつステータスワードの下位バイトにUnknown commandエラーのエラーコード(3)をセットしなければなりません。

デバイスドライバに送られるコマンドブロック

WRITEファンクションコールによりデバイスドライバに送られるすべてのコマンドブロックは，実行する動作を示すバイトで始まります。それに続くバイトは，そのコマンドの付加的な情報を表わします。

ハードウェアブレークポイントをセットするとき，渡されたパラメータブロックが受け入れられるものであるかどうかをチェックすることは，デバイスドライバの責任です。ハードウェアデバッガが実行できない動作を要求するフィールドを，ただ無視して済ませることはできません。各フィールドをチェックして，ハードウェアが要求された動作をサポートするかどうかを確認し，もしサポートしないときは適当なリターンコードをセットしなければなりません。

最初のバイトには，次のコマンドコードの1つを含むことができます。

- ■**0** - Installs vectors(ベクターをセットする)
- ■**1** - Gets hardware capabilities(ハードウェアの機能を取得する)
- ■**2** - Enables the hardware breakpoints(ハードウェアブレークポイントを使用可にする)
- ■**3** - Disables the hardware breakpoints(ハードウェアブレークポイントを使用不可にする)
- ■**4** - Sets a hardware breakpoint(ハードウェアブレークポイントをセットする)
- ■**5** - Clears a hardware breakpoint(ハードウェアブレークポイントを解除する)
- ■**6** - Sets I/O base address, resets hardware(ベースアドレスをセットし，ハードウェアをリセットする)
- ■**7** - Removes vectors(ベクターを除去する)

次のコマンドに対しては，コマンドコードの後に付加的なデータが送られます。

Install vectors(コード0)

4バイト　ベクタールーチンを指すfarポインタ

32ビットのポインタで，最初のワードがオフセットを，2番目のワードがセグメントを示します。ハードウェアブレークポイントが発生したときにデバイスドライバをジャンプさせるアドレスです。このルーチンはまた，デバイスドライ

バが必要とする割り込みベクターもセットする必要があります。Turbo Debuggerは，起動されたときに一度だけこのルーチンを呼び出します。Turbo Debuggerは，ハードウェアデバッガデバイスドライバを使う必要がなくなったときに，Remove vectorsファンクション(コード10)を一度だけ呼び出します。Remove vectorsファンクションは，Install vectorsファンクションにより退避されているベクターを元に戻し，ハードウェアデバッガを使用不可にしなければなりません。

Set a hardware breakpoint(コード4)

1バイト　ブレークポイントのタイプ

0 Read Memory（メモリからの読み込み）
1 Write Memory（メモリへの書き込み）
2 Access Memory（メモリに対する読み込みおよび書き込み）
3 Input I/O（I/O読み込み）
4 Input I/O（I/O書き込み）
5 Both I/O（I/O読み込みおよび書き込み）
6 Fetch Instruction（命令フェッチ）

1バイト　アドレスマッチングモード

0 Match All（すべてのアドレスにマッチ）
1 Equal（指定されたアドレスにマッチ）
2 Unequal（指定されたアドレス以外のアドレスにマッチ）
3 Above（指定されたアドレスより高位のアドレスにマッチ）
4 Below（指定されたアドレスより下位のアドレスにマッチ）
5 Less or Equal（指定されたアドレスまたはそれより下位のアドレスにマッチ）
6 Greater or Equal（指定されたアドレスまたはそれより高位のアドレスにマッチ）
7 Range（指定された範囲内のアドレスにマッチ）
8 Not Range（指定された範囲外のアドレスにマッチ）

4バイト　下限アドレス

32ビットのリニア形式で表現したメモリアドレスです。アドレスマッチングのためのアドレスを指定します。2つのアドレスを指定しなければならないとき

は，小さい方のアドレスを指定します。

4バイト　上限アドレス

32ビットのリニア形式で表現したメモリアドレスです。アドレスマッチングのために2つのアドレスを指定しなければないときに，大きい方のアドレスを指定します。

2バイト　パスカウント

1バイト　データマッチングサイズ

1 = バイト，2 = ワード，4 = ダブルワード

1バイト　マッチするバスサイクルのソース

1　CPU
2　DMA
3　CPUおよびDMA（どちらでもよい）

1バイト　データマッチングモード

0　Match All（すべての値にマッチ）
1　Equal（指定された値にマッチ）
2　Unequal（指定された値以外の値にマッチ）
3　Above（指定された値より高位の値にマッチ）
4　Below（指定された値より下位の値にマッチ）
5　Less or Equal（指定された値またはそれより小さい値にマッチ）
6　Greater or Equal（指定された値またはそれより大きい値にマッチ）
7　Range（指定された範囲内の値にマッチ）
8　Not Range（指定された範囲外の値にマッチ）

4バイト　下限値

データマッチングのための値を指定します。2つの値を指定しなければならないときは，小さい方の値を指定します。データマッチングサイズは，このフィールドの何バイトが意味があるかを指定します。

4バイト　上限値

データマッチングモードで2つの値を指定しなければならないときに，大きい方

の値を指定します。データマッチングサイズは，このフィールドの何バイトが意味があるかを指定します。

4バイト　データマスク

マッチングを調べるデータのビットを指定します。

Clear a hardware breakpoint（コード5）

1バイト　解除しようとするブレークポイントのハンドルです。このハンドルはSet a hardware breakpointコマンド（コード4)によりTurbo Debuggerに与えられたものです。

Set I/O board base address（コード9）

2バイト　ハードウェアデバッガボードのベースアドレスです。

デバイスドライバにより返されるステータスブロック

READファンクションは，デバイスドライバからTurbo Debuggerにステータスブロックを返します。デバイスドライバにコマンドが書き込まれると，その結果を示すために，デバイスドライバの内部にコマンドに応じて各種のステータスブロックがセットされます。どのステータスブロックも，先頭の1バイトは，要求された動作の全般的な結果を示します。その後のバイトは，そのステータスブロックが生成されたコマンドに特有の付加的な情報を返します。先頭バイトで返すことができるステータスコードは次のとおりです。

0 コマンドは正常に実行された。

1 ハンドルが不正である。

2 満杯，これ以上ブレークポイントを設定できない。

3 ブレークポイントが複雑すぎる。
ブレークポイントがセットできなかったことを示します。使用しているハードウェアデバッガは，Turbo Debuggerが要求したバスサイクル，アドレス，データの組み合わせからなるマッチング条件をサポートできません。

4 前のコマンドに起因する制約のためにコマンドを実行することができない。
それ以前の動作のどれかが実行を妨げなければ，そのコマンドは実行できたことを示します。たとえば，使用しているハードウェアではデータマッチの条件としてただ1つの値しか設定できないときに，第1のブレークポイントの他に，マッチするデータの値が異なる第2のブレークポイントを設定しようとしたときに，このような状態が発生します。

5 デバイスドライバがハードウェアボードを見つけることができない。

6 ハードウェアの故障が発生した。

7 不正なコマンドコードがドライバに送られた。

8 ドライバが関数コード0で初期化されていないので，まだ何もできない。

次のコマンドに対しては，ステータスコードバイトの後にさらに付加的なステータス情報を返します。

Get hardware capabilities(コード1)

2バイト　　デバイスドライバインターフェース（この仕様）のバージョン番号

現バージョンの番号は1です。この次のバージョンは，この番号が大きくなるはずです。

2バイト　デバイスドライバソフトウェアのバージョン番号

デバイスドライバのバージョン番号を入れます。このフィールドの番号を使って，Turbo Debuggerに，必要に応じて特別の手段を取らせることができます。

1バイト　このドライバ/ボードの組み合わせがサポートできるハードウェアブレークポイントの最大数

1バイト　コンフィギュレーションビット

ビット	機能
0	CPUアクセスとDMAアクセスを区別できるとき1
1	DMA転送を検出できるとき1
2	データマスク機能があるとき1
3	ハードウェアパスカウンタがあるとき1
4	アドレスだけでなくデータにもマッチング条件が設定できるとき1

1バイト　サポートするブレークポイントの種類（ビットマスク）

ビット	機能
0	Read Memory（メモリからの読み込み）
1	Write Memory（メモリへの書き込み）
2	Access Memory（メモリに対する読み込みおよび書き込み）
3	Input I/O（読み込みI/O）
4	Output I/O（書き込みI/O）
5	Both I/O（読み込みI/Oおよび書き込みI/O）
6	Fetch Instruction（命令フェッチ）

2バイト　サポートされるアドレスマッチモード（ビットマスク）

ビット	機能
0	Match All（すべてのアドレスにマッチ）
1	Equal（指定されたアドレスにマッチ）
2	Unequal（指定されたアドレス以外のアドレスにマッチ）
3	Above（指定されたアドレスより高位のアドレスにマッチ）

4	Below（指定されたアドレスより下位のアドレスにマッチ）
5	Less or Equal（指定されたアドレスまたはそれより下位のアドレスにマッチ）
6	Greater or Equal（指定されたアドレスまたはそれより高位のアドレスにマッチ）
7	Range（指定された範囲内のアドレスにマッチ）
8	Not Range（指定された範囲外のアドレスにマッチ）

2バイト　サポートされるデータマッチモード（ビットマスク）

ビット	機能
0	Match All（すべての値にマッチ）
1	Equal（指定された値にマッチ）
2	Unequal（指定された値以外の値にマッチ）
3	Above（指定された値より高位の値にマッチ）
4	Below（指定された値より下位の値にマッチ）
5	Less or Equal（指定された値またはそれより下位の値にマッチ）
6	Greater or Equal（指定された値またはそれより高位の値にマッチ）
7	Range（指定された範囲内の値にマッチ）
8	Not Range（指定された範囲外の値にマッチ）

1バイト　最大データマッチ長

ハードウェアが行なうことができるデータマッチまたはデータマスクの最大長。そのバイト数に応じて1，2，4のどれかにセットします。

2バイト　ボード上のメモリのサイズ(Kb)

2バイト　トレースバックできる事象の最大数

Set a hardware breakpoint（コード4）

1バイト　Turbo Debuggerがブレークポイントを指定するために将来使うことが予定されているハンドル。デバイスドライバも，ハードウェアブレークポイントが発生した後にTurbo Debuggerを呼び出すときに，このハンドルを使います。このハンドルは，ゼロよりも大きいか，ゼロに等しくなければなりません。負の値（最上位のビットが1）は，ブレークポイントに関してデバイスドライ

バがTurbo Debuggerを呼び出すときの特別の条件を示します。

デバイスドライバからのTurbo Debuggerの呼び出し

ハードウェアボードとデバイスドライバソフトウェアがハードウェアブレークポイントが発生したと判定したときは，コマンドコード0(ハードウェアブレークポイントベクターをセットする)でセットされたベクターにより指し示されるTurbo Debuggerの内部のアドレスに，制御を移さなければなりません。

ベクターアドレスにジャンプするときは，CPUの状態は，プログラムのAXレジスタがスタックにプッシュされ，AHレジスタにエントリコードがロードされる以外は，ハードウェアブレークポイントが発生したときとまったく同じでなければなりません。エントリコードは次のどちらかです。

>=0	トリガされたブレークポイントのハンドル
-1 (FF)	ブレークポイントボタンが押されたことを示す

ハードウェアブレークポイントが発生して制御がTurbo Debuggerに移ると，デバイスドライバには戻りません。

付録G

リモートデバッグ

Turbo Debuggerのリモートデバッグ機能は,他のデバッガにより提供されるものとは異なります。他のデバッガのリモートデバッグでは，通常のデバッグと同じように，デバッガとデバッグされるプログラムが同じシステム上にあり，ただリモートシステムからデバッガを制御するだけにすぎません。これでは，デバッガがロードされた後に残されるメモリよりもユーザプログラムが大きなメモリを必要とする場合に，通常のデバッグと同じ問題が発生します。Turbo Debuggerパッケージの一部として提供されるTDREMOTEプログラムは，Turbo Debuggerを一方のシステムで動作させ，デバッグしているプログラムを他方のシステムで実行することにより，この問題を解決します。

この付録では，1つのシステムに接続したもう1つのシステムを使って，非常に大きなプログラムをデバッグする方法について説明します。

なぜリモートデバッグを使うのか，疑問に思われる人もいるでしょう。リモートデバッグは，たとえば，デバッグしたいプログラムがTurbo Debuggerのもとにロードできないときに使います。デバッグするプログラムをロードしようとしたときに,“Not enough memory to load symbol table”，あるいは“Not enough memory”というメッセージが表示されたときは，リモートデバッグを使うことを考えてください。

システムにEMSメモリが装備されていて，プログラムをデバッグしているときにメモリの大きさが問題になったときは，シンボルテーブルを格納するためにEMSメモリを使っているかどうか確かめてください。通常Turbo DebuggerはEMSメモリを見つけるとそこにシンボルテーブルを格納しますが，インストールユーティリティTDINSTによってその設定が変更されている可能性もあるからです。

Turbo Debuggerのもとでプログラムは正常にロードできても，プログラムが動作するために必要なメモリが残らないこともときどき起こります。この場合にも，リモートデバッグを使うことにより問題が解決します。

リモートデバッグのためのハードウェアのセットアップ

リモートデバッグ機能を使うには，次のものが必要です。

■シリアルポートを備えたシステム

■シリアルポートと，デバッグするプログラムを収容するために必要なメモリスペースおよびディスクスペースを備えたシステム

■2つのシステムを接続するための“ヌルモデム”ケーブル

2つのシステムを接続するためのケーブルは，必らず正しく結線されているものを使用してください。並行に結線されているケーブルは使用できません。少なくとも送信データ線と受信データ線は，交差されていなければなりません（コンピュータの販売店で購入できるはずです）。

ケーブルが準備できたら，それで2つのシステムのシリアルポートを接続してください。リモートリンクのためのハードウェアのセットアップはこれで完了です。

リモートシステムのソフトウェアの準備

リモートデバッグドライバTDREMOTE.EXEをリモートシステムに置いてください。またデバッグしているプログラムが必要とするファイルも，すべてリモートシステムに置いてください。プログラムが必要とするファイルには，たとえばデータファイル，コンフィギュレーションファイル，ヘルプファイルなどがあります。

ファイルをリモートシステムに置くには，ファイルが入っているフロッピィディスクをリモートシステムのドライブに挿入してもよいし，付録Bに説明されているTDRF(Remote File Transfer)ユーティリティを使ってコピーすることもできます。

デバッグしたいプログラムのコピーは，リモートシステムに置いても置かなくてもかまいません。必要なときは，Turbo Debuggerがリモートリンクを通して送るからです。

TDREMOTEの起動

リモートシステム上でTDREMOTEドライバプログラムを起動するときは，カレントディレクトリにしたいディレクトリが，カレントディレクトリになっていることを確認してください。このことは重要です。なぜならTDREMOTEは，自分が起動されたときのカレントディレクトリに，送られてきたプログラムを置くからです。

またTDREMOTEを起動する前に，リモートシステムのシリアルポートが標準ポートあるいは拡張ポートの1または2(COM1，COM2あるいはCOM3)のどれにセットアップされているか確認してください。シリアルポートが標準ポート(COM1)にセットアップされているときは，次のように入力してTDREMOTEを起動してください。

```
TDREMOTE -rp1 -rs2
```

シリアルポートが拡張ポート1(COM2)にセットアップされているときは，次のように入力してTDREMOTEを起動してください。

```
TDREMOTE -rp2 -rs2
```

上の入力例のどちらも，リンクの通信速度を最高速度(40Kbpsまたは31Kbps，CPUのクロック周波数が5/10MHzの場合は40Kbps，8/16MHzの場合は31Kbps)に設定します(IBM PCの場合は-**rs3**としたときに最高速度115Kbpsになります)。ほとんどのセットアップでは最高速度で動作します。最高速度ではうまくいかないときは，速度を遅くする必要があり

ます。後でもっと遅い通信速度でリンクを開始する方法を説明します。

TDREMOTEは，コピーライトメッセージを表示して，リンクのもう一方のシステムでTurbo Debuggerが起動されるのを待っていることを示します。TDREMOTEを停止してDOSに戻りたいときは***STOP***（IBM PCでは***Ctrl-Break***）を押してください。

注意：IBM PCでは，サポートされているのはCOM1およびCOM2だけです。

Turbo Debuggerの起動

Turbo Debuggerを起動するには，DOSからTurbo Debuggerを起動するときのコマンドラインに，さらに次のオプションを追加してください。

- ■シリアルポートが標準ポート（COM1）のとき：　**-rs2**
- ■シリアルポートが拡張ポート1（COM2）のとき：**-rp2 -rs2**
- ■シリアルポートが拡張ポート2（COM3）のとき：**-rp3 -rs2**

リンクが確立されたときは，リモートシステムに“Turbo Debugger online”というメッセージが表示され，Turbo Debuggerの画面に“TDREMOTE online”というメッセージが表示されます。Turbo Debuggerの画面のメッセージは，すぐに通常のウィンドウの表示に変わります。

TDとTDREMOTEは，（**-rs**オプションによって）同じ通信速度に設定しなければなりません（TDとTDREMOTEは，通信速度およびシリアルポートを指定するために，どちらも同じコマンドラインオプションを使用していることに注目してください）。

Turbo Debuggerには，**-r**コマンドラインオプションがあります。このオプションは，デフォルトの通信速度とシリアルポートを使ってリモートリンクを開始することを指定します。TDINSTを使って変更していなければ，シリアルポートのデフォルトは標準ポート（COM1），通信速度のデフォルトは通信速度の最大値である40/31Kbps（IBM PCでは115Kbps）です。

リモートリンクを確立するための代表的なTurbo Debuggerのコマンドラインは，次のようになります。

```
td -rs2 myprog
```

このコマンドラインは，デフォルトのシリアルポートを通して最高の通信速度でリンクし，*myprog*がまだリモートシステムにロードされていないときは，*myprog*をリモートシス

テムにロードします。

リモートシステムへのプログラムのロード

Turbo Debuggerは，ローカルシステムおよびリモートシステムのプログラムのコピーの日付と時刻を調べます。ローカルシステムのコピーがリモートシステムのコピーよりも新しいときは，Turbo Debuggerは，ローカルシステムのプログラムが，再コンパイルして再リンクされたか，または再リンクされたものとみなし，リンクを通してそのコピーをリモートシステムに送ります。

この転送の速度は，最高通信速度で4～5Kバイト/秒（IBM PCでは約11Kバイト/秒）です。これは60Kバイトのプログラムの転送に，12～15秒（IBM PCでは約6秒）かかることを意味します。したがって，何かプログラムをロードしたいときにすぐにロードされなくても心配しないでください。

何かが進行中であることを示すために，リモートシステムの画面には，転送中のファイルの転送バイト数が表示されます。

TDREMOTEのコマンドラインオプション

TDREMOTEがサポートするコマンドラインオプションは次のとおりです。オプションはハイフン(-)またはスラッシュ(/)のどちらかで始めます。

-?	ヘルプ画面を表示する
-h	ヘルプ画面を表示する
-rs1	低速（9600bps）
-rs2	PC-9801：高速（40/31Kbps）
	IBM PC：中速（40Kbps）
-rs3	高速（115Kbps，IBM PCのみ）
-rp1	標準ポート（COM1）
-rp2	拡張ポート1（COM2）
-rp3	拡張ポート2（COM3，PC-9801のみ）
-w	オプションを書き込む

コマンドラインオプションを指定しないでTDREMOTEを起動すると，TDREMOTEはデフォルトのポートと通信速度を使います。デフォルトは，**-w**オプションを使って変更し

ていなければ，標準ポート（COM1）と40/31Kbps（IBM PCでは115Kbps）です。

-**w**オプションを使うと，コマンドラインオプションをディスク上のTDREMOTEの実行可能なプログラムイメージに書き込んで，恒久的なものにすることもできます。-**w**オプションを指定したときは，実行可能なプログラムファイル名の入力を求めるプロンプトが現われます。

DOSのバージョン3.0またはそれ以降のバージョンを使用しているときは，プロンプトにTDREMOTEを起動したパスとファイル名が表示されます。**リターン**を押してこのファイルを指定することもできますし，パスとファイル名を入力して**リターン**を押すことにより別の実行可能ファイルを指定することもできます。指定したファイルが存在しないときは，その名前で新しいTDREMOTEの新しいコピーが作られます。

DOSのバージョン2を使用しているときは，いつでもパスとファイル名を入力しなければなりません。

リモートデバッグセッション

TDREMOTEとTDをリモートモードで起動したら,単一のシステムでデバッグするときとほとんど同じやり方でデバッグすることができます。Turbo Debuggerのコマンドの働きもまったく同じです。リモートデバッグのために新しく習得しなければならないことは何もありません。

リモートシステムでは,デバッグしているプログラムが実際に実行されるので,画面出力が発生したり,キーボード入力が必要になったりすることを忘れないでください。リモートデバッグを行なっているときは,**F10**/View/User screenコマンドは無効です。

またCPUウィンドウのタイトルの一部に,リモートシステムのCPUの型名が表示されます。CPUの型名の前には,リモートシステムのCPUであることを示すために"REMOTE"と表示されます。

Turbo Debuggerの内部からリモートシステムにファイルを転送したいときは,**F10**/File/OS shellコマンドを使ってDOSに移り,TDRFユーティリティを使ってファイルメンテナンス操作を行ないます。ファイルの転送が終了したら,DOSプロンプトのもとでexitと入力して**リターン**を押すとTurbo Debuggerに戻ることができます。TDRFの使い方は,付録Bに説明されています。

TDREMOTEのメッセージ

TDREMOTEにより表示されるメッセージは次のとおりです。

***nn* bytes downloaded**

***nn*バイトダウンロードされました。**

リモートシステムにファイルが転送されています。このメッセージは,ファイル転送の進行状況を示します。転送速度は,最高リンク速度40/31Kbps(115Kbps)で12～15Kバイト/秒(IBM PCでは約10K/秒)です。

Can't create file

ファイルを作成できません。

TDREMOTEが,送られてくるファイルを書き込むファイルを作成できません。このエラーは,ディスクが一杯のとき,または同名のディレクトリが存在するときに発生

します。

Can't modify exe file

実行可能ファイルを変更できません。

コマンドラインオプションを書き込むために指定した実行可能ファイルが，TDREMOTEユーティリティの適正なコピーではありません。-wオプションを使ってコマンドラインオプションを書き込めるのは，TDREMOTEユーティリティのコピーだけです。

Can't open exe file to modify

変更すべき実行可能ファイルをオープンできません。

変更する実行可能ファイルがオープンできません。不正なファイル名を入力したか，存在しないファイル名を入力した可能性があります。

Download complete

ダウンロードが完了しました。

ファイルがTDREMOTEに正常に送られました。

Download failed, write error on disk

ダウンロードが失敗しました。ディスクへの書き込みエラーです。

TDREMOTEが，受信したファイルの一部をディスクに書き込めません。このエラーは，通常ディスクが一杯になったときに発生します。ファイルをいくつか削除してから，もう一度ダウンロードしてください。

Enter program file name to modify

変更するプログラムファイル名を指定してください。

DOSのバージョン3.0またはそれ以降のバージョンを使用しているときは，プロンプトにTDREMOTEを起動したパスとファイル名が表示されます。**リターン**を押してこのファイルを指定することもできますし，パスとファイル名を入力して**リターン**を押すことにより，別の実行可能ファイルを指定することもできます。指定したファイルが存在しないときは，その名前で新しい実行可能ファイルが作成されます。

DOSのバージョン2を使用しているときは，いつでもパスとファイル名を入力しなければなりません。

Interrupted

中断されました。

他方のシステムとの通信が確立するのを待っている間に，***STOP***キー（***Ctrl-Break***）を押したことを示します。

Invalid command line option

無効なコマンドラインオプションがあります。

DOSのコマンドラインからTDREMOTEを起動するときに，無効なコマンドラインオプションを指定しました。

Link broken

リンクが切断されました。

TDREMOTEと通信しているプログラムが停止し，DOSに戻りました。

Link established

リンクが確立されました。

他方のシステム上のプログラムが，TDREMOTEと通信を開始しました。

Loading program "*name*" from disk

ディスクからプログラム"*name*"をロードしています。

Turbo Debuggerが，TDREMOTEに，デバッグの準備をするためにディスクからメモリにプログラムをロードするように指令しました。

Program load failed, EXEC failure

プログラムのロードに失敗しました。プログラムに問題があります。

DOSがプログラムをメモリにロードできませんでした。このエラーは，プログラムが壊れているか，途中で切れているときに発生します。このメッセージが表示されたときは，ディスクからこのプログラムファイルを削除してください。プログラムファイルを削除することにより，Turbo Debuggerに，リンクを通して新しいコピーを送らせることができます。プログラムファイルを削除してもまだこのメッセージが表示されるときは，他方のシステムのTurbo Debuggerと再リンクして，もう一度やってみてください。

Program load failed; not enough memory

プログラムのロードに失敗しました。メモリが足りません。

リモートシステムに，デバッグしたいプログラムをロードするのに十分な空きメモリがありません。このエラーは，プログラムが非常に大きいとき以外には発生しないは

ずです。なぜならTDREMOTEは，約15Kのメモリしか必要としないからです。

Program load failed; program not found

プログラムのロードに失敗しました。プログラムが見つかりません。

TDREMOTEが，リモートシステムのディスクの中に，プログラムを見つけることができませんでした。このエラーはめったに発生しないはずです。なぜならTDREMOTEがプログラムを見つけることができないときは，Turbo Debuggerが，リモートシステムにそのプログラムをダウンロードするからです。

Program load successful

プログラムが正常にロードされました。

TDREMOTEが，Turbo Debuggerにより指定されたプログラムのロードを終了しました。

Reading file "*name*" from Turbo Debugger

Turbo Debuggerからファイル"*name*"を読み込んでいます。

Turbo Debuggerからファイルが送られています。

Unknown request: *message*

要求がわかりません：*message*

TDREMOTEが他方のシステムから無効な要求を受け取りました。リンクが正常に動作しているときは，このメッセージはめったに表示されないはずです。このメッセージが表示されたときは，接続ケーブルに異常がないか調べてください。ケーブルに異常がないのにこのメッセージが表示されるときは，**-rs**コマンドラインオプションを使って，リンクの通信速度を下げてください。

Waiting for handshake (press stop to quit)

ハンドシェイクを待っています(stopを押すと中断します)

TDREMOTEが起動し，他方のシステムのプログラムが信号を送ってくるのを待っています。他方のシステムのプログラムが通信を開始する前にDOSに戻りたいときは，***STOP***キー（***Ctrl-Break***）を押してください。

システム間のリンクに問題があるとき

リモートデバッグのセットアップで，2つのシステム間のリンクがうまくいかないことがあります。なにか問題が起きたときは，まずケーブルの接続を調べてください。次に**-rs1**コマンドラインオプション（低速）を指定してTDREMOTEとTurbo Debuggerを起動し，リンクしてみてください(IBM PCの場合にはさらに，**-rs1**で正常に動作したら，**-rs2**で試してみてください)。

ハードウェアとケーブルの相性により，高速では正常に動作しないことがあります。低速（または中速）でしか動作しないときは，ケーブルまたはコンピュータを交換してみてください。

付録H

プロンプトとエラーメッセージ

Turbo Debuggerは，現在のカーソル位置に，プロンプトやエラーメッセージを表示します。

この付録には，Turbo Debuggerが表示するすべてのプロンプトとエラーメッセージがアルファベット順に掲載されています。また各プロンプトとエラーメッセージには，その意味と対処方法の説明が付してあります。

プロンプト

Turbo Debuggerは，コマンドを実行するために付加的な情報が必要なときは，その情報の入力を要求するプロンプトを表示します。プロンプトのタイトルは，入力を求める情報を示します。プロンプトの内部には，選択項目や，ヒストリリストのような入力を補助する情報が表示されます。

プロンプトには2つの対応のしかたがあります。

- 要求される応答をする。
- **ESC**を押してプロンプトをキャンセルし，そのプロンプトが表示される直前のコマンドメニューに戻る。

プロンプトには2つの項目(yes/noのような)の選択を求めるだけのものもあります。このようなプロンプトに対しては，矢印キーによりどちらかを選択して**リターン**を押すか，**Y**または**N**を押すか，**ESC**を押してコマンドをキャンセルすることができます。

プロンプトがアクティブなときに使用することができるキーストロークコマンドについては，第2章を参照してください。

次にすべてのプロンプトをアルファベット順に示します。

Already recording, do you want to abort?

すでに記録中です。中止しますか？

キーストロークマクロの記録中です。現在のマクロの記録を終了するかまたは中止するまでは，別のキーストロークマクロの記録を開始することはできません。マクロの記録を中止するときは**Y**を，記録を続行するときは**N**を押してください。

Device error - Retry?

デバイスエラー ― 再試行しますか？

プリンタのようなキャラクタデバイスに書き込んでいるときに，エラーが発生しました。このエラーは，プリンタのケーブルが接続されていないとき，プリンタがオフラインになっているとき，用紙がなくなったときなどに発生します。エラーの原因を取り除いてから，再試行するときは**Y**を，キャンセルするときは**N**を押してください。

Disk error on drive *ddd* - Retry?

ドライブ*ddd*でディスクエラー － 再試行しますか？

ドライブ*ddd*にアクセスしているときに，ハードエラーが発生したことを示します。このエラーの原因は，多くの場合，ドライブにディスクが挿入されていないか，またハードディスクでは，アクセスした部分が読み出しまたは書き込みできないことです。再試行するときは***Y***を，キャンセルしたいときは***N***を押してください。

Edit watch expression

ウォッチ式を編集してください。

式の一部を変更したり，式全体を新しい式に書き換えることができます。プロンプトは，現在ハイライト表示されているウォッチ式に初期化されます。

Enter address, count, byte value

アドレス，バイト数，バイト値を入力してください。

希望するバイト値に，セットしたいメモリブロックの先頭アドレス，ブロックのバイト数，セットしたいバイト値を入力することができます。

Enter animate delay (10th of sec)

Animateの遅延時間を入力してください(1/10秒単位)

Animateコマンドによるトレースの進行速度を指定することができます。指定する数が大きいほど，ステップ(1ソース行または1命令の実行)間の遅延時間が長くなります。

Enter go to offset

移動先のオフセットを入力してください。

ファイルが16進数バイト形式で表示されているときにGotoコマンドを選択すると，このプロンプトメッセージが表示されます。表示させたいデータバイトのファイルの先頭からのオフセットを入力してください。ファイルの内容が，指定したデータバイトから表示されます。

Enter inspect start index, range

調べたい先頭の要素の添字と範囲を入力してください。

配列の中の表示させたい先頭の要素の添字と，表示させたい要素の数を入力してください。要素の添字と個数は，カンマ(,)で区切ってください。

Enter instruction to assemble

アセンブルする命令を入力してください。

Codeペインの中のカレントアドレスのアセンブラ命令と置き換える，アセンブラ命令を入力してください。付録Dには，アセンブリ言語のキーワードの一覧表が掲載されています。またアセンブリ言語のより詳しい説明は，第10章を参照してください。

Enter log file

ログファイル名を入力してください。

ログを書き込みたいファイルの名前を入力してください。Close log fileコマンドを入力するまでは，ログに送られるすべての行は，ディスプレイに表示されるだけでなく，このファイルにも書き込まれます。ファイル名を入力しないで**リターン**を押すと，デフォルトのファイル名が使用されます。デフォルトのファイル名は，デバッグしているプログラムと同じファイル名で，拡張子が.LOGになります。

Enter memory address

メモリアドレスを入力してください。

メモリアドレスを入力してください。シンボル名や式を使うこともできます。

Enter name of configuration file

コンフィギュレーションファイルの名前を入力してください。

読み込みまたは書き込みを行なうコンフィギュレーションファイルの名前を入力してください。コンフィギュレーションファイルから読み込むときは，DOS形式のワイルドカードマスクを使ってマッチするファイルのリストを表示させ，そのリストから選択することもできます。

Enter name of file to view

ビューするファイルの名前を入力してください。

DOS形式のワイルドカードを使ってファイルのリストを表示させ，そのリストから選択することもできます。また直接ファイル名を入力することもできます。

Enter new bytes

新しいバイトを入力してください。

ファイル中のカーソルにより指定した位置のバイトと置き換えるバイトリストを入力してください。バイトリストの説明は第9章を参照してください。

Enter new coprocessor register value

コプロセッサのレジスタにセットする新しい値を入力してください。

数値演算コプロセッサの，現在ハイライト表示されているレジスタにセットする新しい値を入力してください。式を入力して新しい値を生成することもできます。式の値は，レジスタにロードされる前に，適正な浮動小数点数形式に変換されます。

Enter new data bytes

新しいデータバイトを入力してください。

セグメント内の，カーソルにより指定した位置のバイトと置き換えるバイトリストを入力してください。バイトリストの説明は第9章を参照してください。

Enter new directory

新しいディレクトリを入力してください。

カレントディレクトリにしたい新しいディレクトリ名(ドライブ名やパスを含む)を入力してください。

Enter new file mask

新しいファイルマスクを入力してください。

DOS形式のワイルドカードを使って，ファイルリストに表示させたいファイルにマッチするファイル名を指定してください。指定の一部として，ドライブ名やディレクトリパスを入力することができます。

Enter new value

新しい値を入力してください。

現在ハイライト表示されているCPUレジスタにセットする新しい値を入力してください。式を入力して新しい値を生成することもできます。

Enter port number

ポート番号を入力してください。

読み込みたいI/Oポートの番号を入力してください。有効なポート番号は0から65535です。

Enter position address

位置アドレスを入力してください。

表示させたいコードまたはデータがある，プログラム内のアドレスを入力してください。アドレスの入力方法の詳しい説明は，第9章の「**コードアドレス，データアドレス，行番号**」の節を参照してください。

Enter program name to load

ロードするプログラムの名前を入力してください。

デバッグするプログラムの名前を入力してください。DOS形式のワイルドカードを使ってファイルのリストを表示させ，そのリストから選択することもできます。またロードするファイルの名前を入力することもできます。ファイル名に拡張子を付けないときは，.EXEを指定したものとみなされます。

Enter read file name

読み込むファイル名を入力してください。

メモリに読み込みたいファイル名，またはワイルドカードを含む指定を入力してください。ワイルドカード指定を入力するか，またはデフォルトの*.*を受け入れたときは，マッチするファイルのリストが表示され，その中から選択することができます。

Enter run to code address

実行を停止するコードのアドレスを入力してください。

実行を停止したいプログラム内のアドレスを入力してください。アドレスの入力方法の詳しい説明は，第9章の「**コードアドレス，データアドレス，行番号**」の節を参照してください。

Enter search bytes

検索するバイトを入力してください。

検索したいバイトリストを入力してください。検索は，カーソルにより指定されるメモリ位置から開始されます。バイトリストの説明は第9章を参照してください。

Enter search expression

検索する式を入力してください。

検索する文字列を入力してください。正確に記憶していない文字列を指定するときは，DOS形式のワイルドカード機能を使うことができます。

Enter search instruction or bytes

検索する命令またはバイトを入力してください。

命令は，Assembleローカルメニューコマンドに対して入力するのと同じように入力してください。バイトリストは，DataペインのSearchコマンドに対して入力するのと同じように入力してください。

Enter set breakpoint at code address

ブレークポイントをセットするコードアドレスを入力してください。

プログラム内のブレークポイントを設定したいアドレスを入力してください。アドレスの入力方法の詳しい説明は，第9章の「**コードアドレス，データアドレス，行番号**」の節を参照してください。

Enter source address, destination, count

ソースアドレス，デスティネーションアドレス，バイト数を入力してください。

移動させたいメモリブロックのアドレス，移動先のアドレス，移動させるバイト数を入力してください。各式はカンマ(,)で区切ってください。

Enter source directory list

ソースディレクトリのリストを入力してください。

ディレクトリのリストを入力してください。各ディレクトリは，空白またはセミコロン(;)で区切ってください。Turbo Debuggerは，ソースファイルを見つけるために，リストに現われる順にこれらのディレクトリを探します。

Enter tab column spacing

タブ桁の間隔を入力してください。

Turbo DebuggerがFileウィンドウやModuleウィンドウ内にファイルを表示するときの，タブの間隔を入力してください。指定できる数は1から32です。

Enter variable to inspect

調べたい変数を入力してください。

内容を調べたい変数の名前または式(評価結果がアドレスを指すものでなければなりません)を入力してください。Textペイン内で指定されている変数または式がプロンプト内に表示されているときは，**リターン**を押してそれを受け入れるか，部分的に変更するか，まったく別の変数または式を入力することができます。

Enter variable to range inspect

範囲指定で調べたい変数を入力してください。

内容を配列として調べたい変数の名前または式(評価結果がアドレスを指すものでなければなりません)を入力してください。Textペイン内で指定されている変数または式がプロンプト内に表示されているときは，**リターン**を押してそれを受け入れるか，部分的に変更するか，まったく別の変数または式を入力することができます。

Enter watch expression

監視すべき式を入力してください。

Watchウィンドウの中に入れて値を監視したい変数の名前または式を入力してください。アドレスを指す式だけでなく，$x * y + 4$ のようなアドレスを参照しない式も入力できます。Textペイン内で指定されている変数または式がプロンプト内に表示されているときは，**リターン**を押してそれを受け入れるか，部分的に変更するか，まったく別の変数または式を入力することができます。

Enter write file name

書き込むファイルの名前を入力してください。

メモリブロックの内容を書き込むファイルの名前を入力してください。

Overwrite existing macro on selected key?

選択されたキーにすでに設定されているマクロに上書きしますか？

マクロを設定するキーとして指定した(押した)キーに，すでにマクロが設定されていることを示します。すでに設定されているマクロに上書きしたいときは***Y***を，コマンドをキャンセルするときは***N***を押してください。

Pick a module

モジュールを選択してください。

プログラムを構成するすべてのモジュールのリストが提示されます。そのリストからModuleウィンドウに表示したいモジュールの名前を選択してください。プログラムのモジュールでないファイルを表示したいときは，View/Fileコマンドを使ってください。

Pick a source file

ソースファイルを選択してください。

表示されたリストからソースファイルを選択してください。リストには，カレントモジュールが生成されたソースファイルしか表示されません。

Press key for macro assign

マクロを設定するキーを押してください。

マクロを設定したいキーを押してください。次にそのマクロキーに設定したいコマンドシーケンスを入力してください。コマンドは，マクロの設定のために入力するときも，実際に実行されます。マクロの記録シーケンスを終了するには，そのマクロを設定するキーをもう一度押してください。

Press key for macro delete

マクロを削除するキーを押してください。

設定されているマクロを削除したいキーを押してください。このキーは，マクロを設定する前の機能に戻ります。

Program already terminated; reload?

プログラムはすでに終了しました。再ロードしますか？

プログラムが終了した後で，プログラムを実行しようとしました。**Y**を押すとプログラムが再ロードされます。**N**を押すと，プログラムは再ロードされず，プログラムを実行するコマンドもキャンセルされます。

Program out of date on remote; send over link?

リモートシステムのプログラムは最新のものではありません。リンクを通して送りますか？

リモートシステムにデバッグしたいプログラムが存在しないか，またはリモートシステムのプログラムがローカルシステムのプログラムよりも古いことを示します。**Y**を押すと，日付の新しいプログラムがリモートリンクを通してリモートシステムに送られます。**N**を押すと，ロードコマンドはキャンセルされます。リンク速度が最低速度に設定されているときは，ロードコマンドをキャンセルして，フロッピィディスクを使ってプログラムをコピーした方がよいかもしれません。リンク速度が最高速度に設定されているときは，データ転送速度は，少なくともフロッピィディスクを使ってコピーするのと同じくらい高速です。

Reload program so arguments take effect?

引数の効力が生じるようにプログラムを再ロードしますか？

デバッグしているプログラムのコマンドライン引数を変更しました。**Y**を押すとプログラムが再ロードされ，デバッグ開始時の状態にセットされます。コマンドライン引数を変更したときは，通常再ロードしたいでしょう。なぜなら多くのボーランド社の言語で書かれたプログラムは，ロードされるときに一度だけ引数を見るからです。それ以後のプログラム引数の変更には，プログラムが再起動されるまで気がつきません。

エラーメッセージ

Turbo Debuggerは，何か異常なことが発生したときは，エラーメッセージによってそれを知らせます。ユーザが選択したコマンドが実行できないことを知らせることもあれば，何かがユーザの期待どおりにいかなかったことを警告することもあります。

通常は，エラーメッセージを表示するとき，同時にブザーを鳴らすようになっています。Turbo Debuggerを起動するときにコマンドラインオプションで指定するか，またはインストールプログラムTDINSTの設定を変更することにより，ブザーを鳴らさないようにすることもできます。

エラーメッセージが表示されたときは，次に何かキーを押すと，画面からエラーメッセージが消えます。このキーは次のコマンドとしても解釈されます。したがって，メッセージが表示された後で何かをしようとするとメッセージが消えてしまいます。そこで***ESC***を押さなければ，エラーメッセージが消えないようにすることもできます。コマンドラインオプションで一時的に変更することも，インストールプログラムTDINSTでデフォルトの設定を変更することもできます。

致命的エラー

致命的エラーが発生したときはいつでも，Turbo Debuggerは動作を中止してDOSに戻ります。致命的エラーのいくつかは，DOSからTurbo Debuggerを起動しようとしたときに発生します。またTurbo Debuggerを使っているときに，何か回復不可能なことが発生したときに起きるものもあります。どちらの場合にも，問題を解決した後で，DOSプロンプトからデバッガを再起動するしか方法はありません。

Bad configuration file

コンフィギュレーションファイルがおかしい。

コンフィギュレーションファイルの内容が壊れているか，Turbo Debuggerのコンフィギュレーションファイルでないか，旧バージョンのデバッガのコンフィギュレーションファイルであるかのいずれかです。

Could not creat dummy PSP segment

ダミーのPSPセグメントを作成できませんでした。

ロードするプログラムを指定しないでTD386仮想デバッガを起動したとき，ダミープログラムが作成できませんでした。デバッグするプログラムを指定して，もう一度TD386を起動してみてください。

Fatal EMS error

致命的EMSエラーです。

EMSメモリドライバが，回復不可能なエラーのコードを返しました。EMSメモリのハードウェアが故障しているか，ソフトウェアドライバが壊れているかどちらかです。

Invalid switch: *swt*

スイッチ*swt*が無効です。

DOSのコマンドラインに無効なスイッチオプション*swt*を指定しました。付録Aには，コマンドラインスイッチのリストが掲載されています。また各コマンドラインスイッチの詳細な説明は第4章を参照してください。

Not enough memory

メモリが足りません。

Turbo Debuggerがコマンドを処理している間に，作業メモリが足りなくなりました。

Remote link timeout

リモートリンクがタイムアウトしました。

リモートシステムとのリンクが切断されました。両方のシステムを再ブートし，再起動してください。それでも問題が解決しないときは，リモートシステム上でのデバッグについて説明している付録Gを参照してください。

Unsupported video adapter

ビデオアダプタがサポートされません。

Turbo Debuggerは，使用されているディスプレイアダプタの機種を識別できません。サポートするビデオアダプタは，MDA, CGA, EGA, VGA, MCGA, Hercules, Compaq合成ディスプレイ，AT&Tおよびこれらと互換性の高い製品です。

Wrong version of TDREMOTE

TDREMOTEのバージョンが正しくありません。

リモートシステム上のTDREMOTEは，Turbo Debuggerとバージョンが合っていません。Turbo Debuggerと同時に提供されたTDREMOTEを使用してください。

エラーメッセージ

')' expected

')'が必要です。

式を評価しているときに，右カッコが足りませんでした。このエラーは，左カッコから始まった式(または式の一部)がそれに対応する右カッコで終わらないときに発生します。たとえば

```
3 * (7 + 4)
```

が，次のようになっている場合です。

```
3 * (7 + 4
```

':' expected

':'が必要です。

Cの式を評価しているときに，三項演算子(？：)の最初の2つの式を区切る疑問符(?)はありましたが，2番目と3番目の式を区切るコロン(:)が見つかりませんでした。たとえば

```
x < 0 ? 4 : 6
```

が，次のようになっている場合です。

```
x < 0 ? 4 6
```

']' expected

']'が必要です。

式を評価しているときに，配列の添字の左カッコ([)に対応する右カッコ(])が見つかりませんでした。たとえば

```
table [4]
```

が，次のようになっている場合です。

```
table [4
```

このエラーは，組み込みアセンブラを使ってアセンブラ命令をアセンブルするときにも，発生することがあります。ベースレジスタまたはインデックスレジスタを使ったメモリアクセスで，レジスタの名前を囲む右カッコが脱落している場合です。たとえば

```
mov  ax,4[si]
```

が，次のようになっている場合です。

```
mov  ax,4[si
```

Already logging to a file
すでにログをファイルに書き込んでいます。

Open log fileコマンドを使ってファイルをオープンした後，Close log fileコマンドを使ってそのファイルをクローズしないで，再度Open log fileコマンドを発行しました。

Assignment out of range
代入しようとした値が範囲外です。

Pascalで代入を行なうときに，変数に，その変数の取り得る値の範囲外の値を代入しようとしました。

Can't change that symbol
そのシンボルを変更することはできません。

変更できないシンボルを変更しようとしました。直接変更できるシンボルは，スカラー（Cではint，long，その他，Pascalではbyte，integer，longint，その他）とポインタだけです。構造体または配列を変更したいときは，各メンバまたは要素を一度に1つずつ変更しなければなりません。

Can't execute DOS command processor

DOSのコマンドプロセッサを実行できません。

DOSのコマンドプロセッサを実行するのに十分なメモリがないか，コマンドプロセッサが見つからないかのどちらかです。COMSPEC環境変数により，DOSのコマンドプロセッサを探すディレクトリが正しく指定されているかどうか調べてください。

Can't have more than one segment override

セグメントオーバーライドはひとつしか指定できません。

両方のオペランドにセグメントオーバーライドを指定した命令をアセンブルしようとしました。どちらかひとつのオペランドだけにしか，セグメントオーバーライドを指定できません。たとえば

```
mov   es:[bx],ds:ax
```

は，次のようにしなければなりません。

```
mov   es:[bx],ax
```

Can't set a breakpoint at this address

このアドレスにブレークポイントを設定することはできません。

ROM，存在しないメモリ，またはセグメント0内にブレークポイントを設定しようとしました。ROM内のプログラムの実行を調べるただひとつの方法は，**Run/Trace**コマンドを使って1命令ずつ実行することです。

Can't set any more hardware breakpoints

これ以上ハードウェアブレークポイントを設定することはできません。

既設定のハードウェアブレークポイントをどれか解除してからでなければ，新しく設定することはできません。ハードウェアデバッガボードの機種により，サポートされるハードウェアブレークポイントの数とタイプが異なります。

Can't set that sort of hardware breakpoint

そのようなハードウェアブレークポイントは設定できません。

CONFIG.SYSファイルにインストールされているハードウェアデバイスドライバは，指定されたようなサイクルタイプ，アドレスマッチ，データマッチの組み合わせをマッチ条件とするハードウェアブレークポイントをサポートしません。

Can't swap user program to disk

ユーザのプログラムをディスクにスワップすることができません。

発行されたコマンドを実行するには，デバッグしているプログラムをディスクに退避しなければなりません。カレントディスクにはプログラムを書き込む空き領域がありません。プログラムの退避を必要とするコマンドを使用するときは，前もってディスクに空き領域を作っておかなければなりません。File/OS shellメニューコマンドとTextペインのEditコマンドは，どちらもプログラムの退避を必要とします。

Can't use same register twice

同じレジスタを2度使用することはできません。

同一のメモリオペランドで，ベースレジスタまたはインデックスレジスタを2度使用している命令をアセンブルしようとしました。どのオペランドでも，ひとつのレジスタは1度しか使用できません。たとえば

```
mov ax,[bx+bx]
```

は，次のように書かなければなりません。

```
mov ax,[bx+si]
```

Can't access an inactive scope

インアクティブなスコープにアクセスすることはできません。

Moduleウィンドウ内の変数で，現在アクティブな関数の外部にある変数を含む式を入力したか，それらの変数をポイントします。アクティブでない関数の中の変数の値は無意味です。そのような変数を式中に使用したり，それらの値を調べたりすることはできません。

Destination too far away

目的アドレスが遠すぎます。

目的アドレスが遠すぎる条件ジャンプ命令をアセンブルしようとしました。条件ジャンプ命令の目的アドレスは，命令自身のアドレスから±128バイトの範囲内でなければなりません。

Divide by zero

ゼロによる割算です。

入力した式が除算演算子（/, **div**）や剰余演算子（%,**mod**）を含み，これらの演算子の右側の式（除数）の評価値がゼロになります。除算演算子（/, **div**）や剰余演算子（%, **mod**）は，この場合の値がどうなるか定義されていないので，エラーメッセージが表示されます。

Edit program not specified

編集プログラムが指定されていません。

ModuleウィンドウまたはDiskウィンドウからEditローカルメニューコマンドを使おうとしましたが，インストールプログラムを使ってエディタ起動コマンドを指定していません。

Error loading program *prg*

プログラム*prg*をロードしているときにエラーが発生しました。

DOSは，指定されたプログラムをロードすることができませんでした。指定されたファイルが適正な.EXEファイルでないか，内容が壊れている可能性があります。

Error opening file *fname*

ファイル*fname*をオープンしているときエラーが発生しました。

Turbo Debuggerは，Fileウィンドウに表示するように指定されたファイルをオープンできませんでした。

Error opening log file *fname*

ログファイル*fname*をオープンしているときエラーが発生しました。

Log to fileローカルメニューコマンドに指定されたファイル名で，ファイルをオープンできませんでした。ファイルを作成するために必要な空き領域がないか，指定したドライブ，ディレクトリパス，ファイル名が正しくないかのどちらかです。ディスクからどれかのファイルを削除してそのファイルのための空き領域を作るか，正しいディスク，パス，ファイル名を指定してください。

Error reading block into memory

ブロックをメモリに読み込むときにエラーが発生しました。

指定されたブロックを，ファイルからメモリに読み込むことができませんでした。ファイルのバイト数よりも大きなバイト数を指定した可能性があります。

Error recording keystroke macro

キーストロークマクロを保存しているときにエラーが発生しました。

記録されたキーストロークマクロをコンフィギュレーションファイルに書き込んでいるときに，エラーが発生しました。このマクロは，ディスクに保存されていない可能性があります。

Error saving configuration

コンフィギュレーションを保存しているときにエラーが発生しました。

Turbo Debuggerは，コンフィギュレーションファイルをディスクに書き込むことができませんでした。ディスクに空き領域を用意してください。

Error swapping in user program, press key to reload

ユーザプログラムに切り替えているときにエラーが発生しました。再ロードするには何かキーを押してください。

他のプログラムを実行するために，デバッグ中のプログラムをディスクに退避させた後，そのプログラムを再ロードすることができません。プログラムが退避されたディスクファイル(SWAP.$$$)を，誤って削除した可能性があります。デバッガにできることは，File/Loadメニューコマンドが発行されたときとまったく同じやり方で，プログラムを再ロードすることだけです。

Error writing block to disk

ブロックをディスクに書き込んでいるときにエラーが発生しました。

指定されたブロックを，指定されたファイルに書き込むことができませんでした。ディスクの空き領域の大きさを超えるバイト数を指定した可能性があります。

Error writing to file

ファイルに書き込んでいるときにエラーが発生しました。

変更をファイルに書き戻すことができませんでした。ファイルが読み出し専用ファイルであるか，またはディスクに書き込んでいるときにハードウェアエラーが発生した可能性があります。

Expression accesses more than one scope

式が複数のスコープにアクセスします。

ブレークポイントとの組み合わせで入力した式に，アクセスできないスコープ内の変数の参照があります。Pascalではローカルな変数およびパラメータ，グローバル変数，外部のサブプログラムのローカル変数（そのブレークポイントが，ネストされた手続きまたは関数の内部にあるとき）を参照できます。Cでは関数の自動変数，モジュールの静的変数，プログラムのグローバル変数を参照できます。ただし，1つの関数の自動変数しか参照できません。

Expression too complex

式が複雑すぎます。

入力された式は複雑すぎます。演算子とオペランドの数がもっと少ない式を入力しなければなりません。1つの式に入れることができる演算子とオペランドの数は64までです。オペランドは定数や変数などです。演算子は，たとえば加算演算子(+)，代入演算子(=，:=)，構造体メンバアクセス演算子(->)，集合要素演算子(**in**)などです。

Expression with side effects not permitted

副作用のある式は使えません。

入力した式は，評価されるときにメモリアドレスの内容を変更します。Turbo Debuggerが繰り返し式を評価しなければならない場合(たとえばWatchesウィンドウ内に表示されるとき)は，このような式は入力できません。

Extra input after expression

式の後に余分な入力があります。

入力した式の形式は正しいのですが，式の後に余分なテキストがあります。式中の演算子が脱落しているときに発生します。たとえば

```
3 * 4 + 5/2                add  ax, 45
```

が，次のようになっている場合です。

```
3 * 4 + 5 2                add  ax, 4 5
```

Help file *fname* not found

ヘルプファイル*fname*が見つかりません。

ヘルプ画面を含むディスクファイルが見つかりませんでした。ヘルプファイルがデバッガと同じディレクトリに登録されているかどうか調べてください。

Immediate operand out of range

即値オペランドが範囲外です。

オペランドに,バイトサイズのオペランドと1バイトに入りきらない即値オペランドが混在する命令を入力しました。たとえば

```
add BYTE PTR[bx],300
```

は，次のようにしなければなりません。

```
add WORD PTR[bx],300
```

Initialization not complete

初期化が終了していません。

コンパイラの初期化コードによってデータセグメントが正しくセットアップされる前に，プログラムが変数をアクセスしようとしました。ほとんどのプログラム変数は，ソースコードの先頭までコンパイラの初期化コードを実行した後でなければ，プログラム変数をアクセスできません。

Invalid argument list

引数リストに誤りがあります。

入力した式の中に，引数リストの書式に誤りがある手続き呼び出しまたは関数呼び出しが含まれています。

引数リストは，左カッコで始まり，ゼロまたは複数個のカンマで区切られた引数(式)が並び，右カッコで終わります。引数のない関数や手続きを呼び出すときも，空のカッコが必要なことに注意してください。

このエラーは，たとえば

```
myfunc(1,2,3)
```

が，

```
myfunc(1,2 3)
```

のようになっている場合に発生します。

Invalid cast syntax

キャストの構文に誤りがあります。

入力した式中に，構文に誤りのある型キャストが含まれています。

Cの型キャストの正しい構文は，次に示すように左カッコで始まり，データ型の宣言を含み(変数名は除く)，右カッコで終わります。たとえば次のようになります。

```
(struct *) x
```

またPascalの型キャストの正しい構文は，既知のデータ型，左カッコ，式，右カッコの順です。たとえば次のようになります。

```
longint(p)                word(p^)
```

Invalid character constant

文字定数の書式に誤りがあります。

入力した式中に書式が正しくない文字定数があります。文字定数は，1文字を単引用符(')で囲んだものです。たとえば

```
x = 'a'
```

が，

```
x = 'a
```

のようになっている場合にこのエラーが発生します。

Invalid far address

farアドレスの書式に誤りがあります。

アセンブルするために入力したJMP命令またはCALL命令の，ジャンプ先アドレスとして指定したfarアドレスの書式が間違っています。farアドレスは2つの4桁の16進数をコロン(:)で区切ったものです。たとえば

```
JMP 1234:XYZ
```

は，次のようにしなければなりません。

```
JMP 1234:1000
```

Invalid format string

書式文字列に誤りがあります。

式の後に書式文字列を入力しましたが，書式文字列が間違っています。書式文字列の詳しい説明は第9章を参照してください。

Invalid function parameter

関数のパラメータに誤りがあります。

式中で関数を呼び出そうとしましたが，その関数呼び出しのパラメータの指定が間違っています。

Invalid instruction

命令に誤りがあります。

入力した命令のニーモニックは正しいのですが，指定したオペランドが正しくありません。通常このエラーは，**POP** CS命令をアセンブルしようとしたときに発生します。

Invalid instruction mnemonic

命令ニーモニックに誤りがあります。

アセンブルする命令を入力するときに，命令のニーモニックを入力しませんでした。命令は，命令ニーモニックとそれに続くオプションのオペランドから構成されます。たとえば

```
mov ax,123
```

を，

```
ax,123
```

と入力した場合などに，このエラーが発生します。

Invalid operand separator

オペランドの区切り文字に誤りがあります。

アセンブルする命令を入力しましたが，オペランドをカンマで区切りませんでした。命令に複数のオペランドがある場合は，いつでもオペランドをカンマで区切らなくてはなりません。たとえば

```
ADD ax 12
```

は，次のようにしなければなりません。

```
ADD ax,12
```

Invalid operand(s)

オペランドに誤りがあります。

アセンブルしようとした命令に，ひとつまたは複数の許されないオペランドが指定されています。たとえば，**MOV**命令に2つのメモリ参照オペランドを指定することはできません。また命令の中には，ワードサイズのオペランドしか操作できないものもあります。たとえば

```
POP al
```

は，次のようにしなければなりません。

```
POP ax
```

Invalid operator/data combination

演算子とデータの組み合わせに間違いがあります。

式の中に，演算を実行できない演算子とオペランドの組み合わせがあります。たとえば，定数にプログラム中の関数のアドレスを掛けようとした場合などです。

Invalid pass count entered

無効なパスカウントを入力しました。

1～65535の範囲にないブレークポイントのパスカウントを入力しました。パスカウントは0にセットできません。パスカウント1は，ブレークポイントが最初に検出されたときにトリガされることを意味します。

Invalid register

レジスタの指定に誤りがあります。

アセンブルする命令の一部として間違った浮動小数点レジスタを指定しました。浮動小数点レジスタは，文字STと，カッコで囲んだ0～7の数字(指定してもしなくてもよい）により指定します。たとえばST，ST(4)のように指定します。

Invalid register combination in address expression

アドレス式中に不正なレジスタの組み合わせがあります。

入力した命令に，ベースレジスタとインデックスレジスタの許された組み合わせ以外の組み合わせを含むオペランドを指定しました。アドレス式は，ベースレジスタだけ，インデックスレジスタだけ，またはベースレジスタとインデックスレジスタをひとつずつ含むことができます。ベースレジスタはBPとBX，インデックスレジスタはSIとDIです。次に適正なアドレスレジスタの組み合わせを示します。

```
BX   BX+SI
BP   BP+SI
DI   BX+DI
SI   BP+DI
```

Invalid register in address expression

アドレス式中に不正なレジスタがあります。

入力した命令の大カッコ([ ])で囲んだメモリアドレス式の中に，間違ったレジスタを使おうとしました。アドレス式には，BX，BP，SI，DIレジスタしか使えません。

Invalid symbol in operand

オペランド中のシンボルに誤りがあります。

アセンブルする命令を入力するときに，オペランドの先頭に，オペランドの先頭に使用してはいけない文字，たとえばコロン(：)などを使用しました。

Invalid value entered

指定した値が無効です。

メモリアドレスの入力を要求するプロンプトに対して，整数値のかわりに浮動小数点数値を入力しました。

Keyword not a symbol

シンボルでなくキーワードです。

(Cとアセンブラのみ) 入力した式は, 変数名が入るべき場所にキーワードが入っています。特殊な演算子**sizeof**以外のキーワードは, 型キャスト演算の一部として使用するとき以外, 式の中には使用できません。たとえば

```
floatval = (char) charval
```

が,

```
floatval = char charval
```

となっている場合などに, このエラーが発生します。

Left side not a structure or union

左側が構造体でも共用体でもありません。

Cの構造体のメンバアクセス演算子 (.または->) またはPascalのレコードフィールド修飾子(.)を使用する式を入力しましたが, これらのシンボルの前(左側)に構造体やレコードの名前, または構造体やレコードを指すポインタがありません。

No coprocessor or emulator installed

コプロセッサまたはエミュレータが組み込まれていません。

メインメニューバーのView/Numeric processorコマンドを使ってNumeric Processorウィンドウを開こうとしましたが, システムに数値演算用のプロセッサが付いておらず, またデバッグしているプログラムがソフトウェアエミュレータを使用していません。

No hardware debugging available

ハードウェアデバッグは利用できません。

ハードウェアデバッガの支援を必要とするブレークポイントを設定しようとしましたが, ハードウェアデバッガのデバイスドライバがインストールされていません。ハードウェアデバッガのデバイスドライバが必要なハードウェアボードを見つけることができないときも, このエラーメッセージが表示されます。

No help for this context

現在の状況のためのヘルプ画面がありません。

ヘルプ画面を表示させるために***F1***キーを押しましたが，現在の状況でどうしたらよいかを示す必要なヘルプ画面を見つけることができませんでした。MSAカスタマサポートセンターに連絡してください。

No modules with line number information

行番号情報の付いたモジュールがありません。

View/Moduleコマンドを発行しましたが，Turbo Debuggerは必要なデバッグ情報の付いたモジュールを見つけることができません。デバッグ情報のついたモジュールがないと，Moduleウィンドウでソースモジュールを見ることができません。このエラーは，通常シンボルテーブルのないプログラムのデバッグ中に発生します。シンボルテーブルの詳しい説明は，エラーメッセージ“**Program has no symbol table**”の説明を参照してください。

No previous search expression

検索対象が指定されていません。

Textペインのローカルメニューから**Next**コマンドを実行しようとしましたが，その前に**Search**コマンドで検索対象を指定していませんでした。**Next**コマンドは，同じペインで**Search**コマンドを発行した後でなければ使えません。

No program loaded

プログラムがロードされていません。

プログラムがロードされていることを必要とするコマンドを発行しました。このようなコマンドはたくさんあります。たとえば**Run**メニュー内のコマンドは，どれもプログラムがロードされていなければ実行できません。このようなコマンドを使用するときは，その前に，**File/Load**コマンドを使ってプログラムをロードしてください。

No source file for module ***mmm***

モジュール*mmm*のソースファイルがありません。

指定されたモジュールのソースファイルが見つかりません。ソースファイルがカレントディレクトリにないときは，Options/Path for sourceコマンドを使って，ソースファイルが存在するディレクトリを指定することができます。

No type information for this symbol

このシンボルの型情報がありません。

デバッグ情報の付いていないプログラム変数を含む式を入力しました。このエラーは，デバッグ情報を生成しないでモジュールをコンパイルした場合に発生します。変数名の前に型キャスト式を付けて変数のデータ型を指定すれば，型情報を補うことができます。

Not a function name

関数名ではありません。

関数呼び出しを含む式を入力しましたが，左カッコの前の名前が関数名ではありません。名前の直後にカッコが付いているときは，式パーサはそれを関数呼び出しと見なし，その名前の関数が見つからない場合はこのエラーメッセージを表示します。

Not a memory referencing expression

メモリを参照する式ではありません。

メモリアドレスを参照しない式を入力しました。単に値を返す式ではなく，メモリアドレスを参照する式を入力しなければならない場合が多くあります。たとえばDataメニューのInspectコマンドは，メモリ内のデータ項目を参照する式の入力を求めます。たとえば

```
3 * 4 < (9 - 1)
```

はメモリアドレスを参照しませんが，

```
myarray[4]
```

はメモリアドレスを参照します。

Not a structure member

構造体のメンバではありません。

Cの構造体のメンバアクセス演算子(.または->)またはPascalのレコードフィールド修飾子(.)のいずれかを使う式を入力しましたが，これらのシンボルの前に構造体やレコードの名前，または構造体やレコードを指すポインタがありません。

Not enough memory for selected operation

メモリ不足のため選択されたコマンドが実行できません。

選択されたコマンドを実行するには新しいウィンドウを開かなければなりませんが，メモリ不足のため新しいウィンドウを開くことができません。どれかのウィンドウを閉じるか，または大きさを縮小してから，このコマンドをもう一度選択してください。

Not enough memory to load program

メモリ不足のためプログラムをロードできません。

プログラムのシンボルテーブルはメモリにロードできましたが，メモリ不足のためプログラムをロードすることができません。システムにEMSメモリが装備されているときは，シンボルテーブルをEMSメモリに格納するように設定されているかどうか調べてください。EMSメモリに格納するようになっていない場合は，**-se**コマンドラインオプションを使ってEMSメモリに格納するように指定するか，TDINSTを使ってデフォルトの設定をEMSメモリに格納するように変更してください。またPC-9801では，同様な方法でシンボルテーブルをGVRAMのバンク＃1に格納するようにセットすることもできます。

システムにEMSメモリが取り付けられていないとき，またはシンボルテーブルをEMSに格納してもプログラムをロードできないときは，2つのシステムを接続して一方でTurbo Debuggerを動かし，他方でデバッグするプログラムを実行するリモートデバッグを使うことができます。リモートデバッグの詳しい説明は付録Gを参照してください。

Not enough memory to load symbol table

メモリ不足のためシンボルテーブルをロードできません。

デバッグするプログラムのシンボルテーブルをメモリにロードするための空き領域がありません。シンボルテーブルには，Turbo Debuggerがソースコードやプログラム変数を表示するときに使う情報が入っています。メモリ上に常駐ユーティリティがある場合はそれらの領域を解放してからデバッガを再起動してください。またコンパイルするときに，デバッグするモジュールとそれに関係するモジュールに対してだけデバッグ情報を生成することにより，シンボルテーブルを小さくして試してみることもできます。

このメッセージが表示されたときは，プログラムはまだロードされていません。したがって，シンボルテーブルやプログラムをロードするのに十分な空き領域を用意する必要があります。

Only one operand size allowed

指定できるオペランドサイズは1つだけです。

2つ以上のサイズ指定子を含む命令を入力しました。オペランドのサイズは，いったんセットしたらそれを変更することはできません。たとえば

```
mov  WORD PTR BYTE PTR[bx],1
```

は，次のようにしなければなりません。

```
mov  BYTE PTR[bx],1
```

Operand must be memory location

オペランドはメモリアドレスを参照しなければなりません。

式中のメモリを参照しなければならないオペランドに，メモリを参照しないオペランドが指定されています。メモリを参照するオペランドを指定しなければならない演算子は，たとえば代入演算子(=, +=など)，インクリメント演算子(++)，デクリメント演算子(--)などです。

Operand size unknown

オペランドのサイズが不明です。

アセンブルする命令を入力しましたが，オペランドのサイズを指定しませんでした。バイトサイズのオペランドおよびワードサイズのオペランドのどちらでも取ることできる命令の中には，オペランドからサイズを判断することができない場合に，どちらのサイズを使用するか指定しなければならないものがあります。たとえば

```
add [bx],1
```

は，次のように指定しなければなりません。

```
add BYTE PTR[bx],1
```

Overlay not loaded

オーバレイプログラムがロードされていません。

(Pascalのみ)CPUウィンドウのペインを，現在メモリにロードされていないプログラム内のロケーションにセットしようとしました。このような場合は，Moduleウィンドウを使うと，そのソースコード(まだメモリにロードされていない)を調べることができます。ただしプログラムがメモリにロードされるまでは，逆アセンブルした命令は見ることができません。

Path not found

パスが見つかりません。

入力されたドライブ名とディレクトリ名の組み合わせが存在しません。正しいドライブを指定したかどうか，ディレクトリパスのスペルに誤りがないかどうか調べてください。

Path or file not found

パスまたはファイルが見つかりません。

ロードするファイル名の入力を求めるプロンプトに対して，存在しないファイル名(パスを含む)または不正なファイル名(パスを含む)を指定しました。ロードしたいファイルの正しい名前がわからないときは，プロンプトが現われたときに**リターン**を押してファイルのリストを表示させ，その中から選択してください。末尾に円記号¥(IBM PCではバックスラッシュ＼)が付いている名前はディレクトリを示します。リストからディレクトリを選択することにより，ディレクトリツリー(木構造)を上または下に移動することができます。

Press Esc

***ESC*キーを押してください。**

このメッセージは，TDINSTにより***ESC***キーを押したときにエラーメッセージが消えるように設定されている場合に，エラーメッセージの一部として表示されます。***ESC***キーを押すとエラーメッセージが消えます。

Program has invalid symbol table

プログラムのシンボルテーブルが無効です。

プログラムの末尾に連結されているシンボルテーブルが壊れています。.EXEファイルをもう一度作成し直して，再ロードしてください。

Program has no symbol table

プログラムにシンボルテーブルが付いていません。

デバッグするプログラムは正常にロードされましたが，デバッグシンボル情報がありません。

CPUウィンドウを使って，機械語レベルでプログラムをデバッグすることができますが，名前でコードやデータを参照することはできません。

Turbo Pascal(5.0以上のバージョン)でシンボルテーブルを作成する場合は，**Debug/Standalone debugging**をオンにしてください（TPC.EXEでは/**v**コマンドラインオプションを使ってください)。Turbo CまたはTurbo Assemblerを使用しているときは，デバッグシンボル情報を連結するために/**v**オプションを指定して，TLINKでリンクしてください。

Program linked with wrong linker version

最新バージョンでないリンカでプログラムをリンクしました。

旧バージョンのデバッグ情報を使ってプログラムをデバッグしようとしました。最新バージョンのリンカを使ってプログラムを再リンクしてください。または最新バージョンのTurbo Pascalを使って再コンパイルしてください。

Program not found

プログラムが見つかりません。

指定されたプログラムが見つかりません。正しいプログラム名を入力するか，ファイルリストからプログラムを選択してください。

Register cannot be used as negative address

レジスタはマイナスのアドレスとして使うことはできません。

ベースレジスタまたはインデックスレジスタをマイナスのオフセットとして使うアセンブラ命令を入力しました。これらのレジスタはプラスのオフセットとしてのみ使用できます。たとえば

```
INC WORD PTR[12-BX]
```

は，次のように指定しなければなりません。

```
INC WORD PTR[12+BX]
```

Register or displacement expected

レジスタまたはオフセットでなければなりません。

入力した命令のオペランドの，大カッコ([ ])内の式に誤りがあります。大カッコ内の式にはレジスタ名と定数のオフセットだけを使うことができ，セグメントベースからのインデックスを示すオペランドになります。

Repeat count not allowed

繰り返し回数を指定することはできません。

繰り返し回数を含む書式文字列を入力しましたが，その書式文字列を適用しようとする式には繰り返し回数を指定できません。

Run out of space for keystroke macros

キーストロークマクロの登録スペースが一杯です。

現在のマクロの記録中に，マクロの登録スペースが満杯になりました。マクロの登録スペースは，全体で256キーストロークです。

Search expression not found

検索式が見つかりません。

指定された文字列またはバイト列が見つかりません。検索は，ファイルのカレントロケーション(カーソルの位置で指定)から始まりファイルの最後で終了します。ファイルの先頭から最後まで検索したい場合は，***CTRL-ROLLDOWN***キー（***Ctrl-PgUp***）を押してカーソルをファイルの先頭に移動してから検索してください。

Source file *fname* not found

ソースファイル*fname*が見つかりません。

指定されたモジュールのソースファイルが見つかりません。このメッセージを表示する前にTurbo Debuggerは次のディレクトリを調べます。

■コンパイラがソースファイルを見つけたディレクトリ

■**-sd**コマンドラインオプションおよびOptions/Path for sourceコマンドで指定されたディレクトリ

■カレントディレクトリ

■デバッガがデバッグ中のプログラムを見つけたディレクトリ

Options/Path for sourceコマンドを使って，ソースファイルが入っているディレクトリを，ディレクトリ検索リストに加えてください。

Symbol not found

シンボルが見つかりません。

無効な変数名を含む式を入力しました。変数名の入力を間違えたか，アクティブでない手続きや関数，または別のモジュールにあるスコープ外の変数を指定した可能性があります。

Syntax error

構文エラーです。

入力した式の構文に誤りがあります。このメッセージよりもっと具体的なメッセージがない場合に表示される，一般的なエラーメッセージです。

Too many files match wildcard mask

ワイルドカードマスクにマッチするファイルが多すぎます。

該当するファイルの数が100を超えるワイルドカードマスクを入力しました。先頭の100個のファイルが表示されます。

Type EXIT to return to Turbo Debugger

EXITとタイプしてTurbo Debuggerに戻ってください。

File/OS shellコマンドを実行しました。DOSのコマンドを実行した後Turbo Debuggerに戻るにはEXITと入力してください。

Unexpected end of line

行が終わるべきでないところで終わっています。

式を評価している途中で式の終わりに達してしまいました。たとえば

```
99 - 22 * 4          SUB AX,4
```

が，次のようなっている場合です。

```
99 - 22 *            SUB AX,
```

Unknown character

識別できない文字があります。

式中に使用できない文字を含む式を入力しました。たとえば，Cでは逆引用符("`"コード値0x60）などです。

Unknown record or structure name

識別できないレコード名または構造体名があります。

式の中に識別できないレコード名，構造体名，共用体名，**enum**名があります（Cの構造体とアセンブラの構造体には，変数名とは別にそれ自身の名前があることに注意してください）。

Unknown symbol

識別できないシンボルがあります。

無効なローカル変数名を含む式を入力しました。モジュール名が間違っているか，そのローカルシンボル名または行番号が間違っているかどちらかです。

Unterminated string

文字列が終了されていません。

入力した文字列を終了する引用符（Cでは"，Pascalでは'）がありません。Pascalでは，引用符を含む文字列を入力するときは引用符(')を1つ余分に入力しなければなりません。Cでは，引用符を含む文字列を入力するときは，引用符の前に円記号¥(IBM PCではバックスラッシュ＼）を付けなければなりません。

Value must be between 1 and 32

値は1から32でなければなりません。

タブ間隔に無効な値を指定しました。タブ間隔として指定できる値は1から32の範囲です。

Video mode not available

指定したビデオモードは使えません。

43/50行モードに切り換えようとしましたが，使用しているディスプレイアダプタはこのモードをサポートしません。

情報メッセージ

Turbo Debuggerは，通常のウィンドウ表示を開始する前に情報メッセージをいくつか表示します。次にこれらの情報メッセージについて説明します。

TDREMOTE online

TDREMOTEオンライン

Turbo Debuggerは，リモートシステムのTDREMOTEリモートデバッグドライバプログラムと通信リンクを確立しました。以後DOSコマンドラインでプログラムの名前を入力すると，そのファイルはリモートシステムにロードされます。

Waiting for handshake from TDREMOTE (*STOP* to quit)

TDREMOTEからのハンドシェイクを待っています(中止する場合は*STOP*を押してください)

Turbo Debuggerに対して，シリアルポートを通して接続されたリモートシステム上のプログラムをデバッグするように指示しました。デバッガは，リモートシステムからデバッグするプログラムが起動したことを知らせてくるのを待っています。

Turbo Debuggerを停止してDOSプロンプトに戻りたいときは，***STOP*** (***Ctrl-Break***)を押してください。

付録 I

Turbo Debuggerをいろいろな言語に使用する

この付録Iには，いろいろな言語に対してTurbo Debuggerを効果的に使用するためのアドバイスを集めてあります。

Turbo Cのための助言

コンパイラによるコードの最適化

TCC(コマンドライン版)に対して-Oコマンドラインオプションを指定するか，TC(統合開発環境版)でOptions/Compiler/Optimizationコマンドを選択して，コードの最適化を指定したときは，ソースコードのところどころで1行ずつ実行することが困難になることがあります。特に複数の**if/else**文があるとき，または**if/else**文がネストされているときは，各**else**節で停止させることが困難なこともあります。また**for**ループも，ループをトレースするとき，少し意外な動作をすることがあります。

この問題はそれほど頻繁には発生しません。また問題が起きたときにCPUウィンドウを開いてアセンブラレベルのデバッグに切り替えるか，コンパイルするときに最適化を抑制することにより回避することができます。

ポインタデータへのアクセス

Cでは，データの配列を参照するために，ポインタを非常によく使います。ポインタ変数を監視すると，通常Turbo Debuggerは，ポインタにより指し示されるひとつの項だけを示します。配列としてのポインタにアクセスするには，まず通常の手法のひとつ，たとえばModuleウィンドウ内の変数の上にカーソルを移動して***CTRL-I***を押すことにより配列を表示させ，次にInspectorウィンドウのローカルメニューのRangeコマンドを使って配列の範囲を指定します。たとえば，プログラムの中に次のようなコードが含まれているときは，

```
char *p, buf[80];
for (p = buf; p < buf + sizeof(buf); p++) {
    ...
}
```

Inspectorウィンドウのローカルメニューの**Range**コマンドを選択し，要素の添字0と要素の数80を入力することにより，*p*を文字の配列として調べることができます。

複雑な式の部分式の値を調べる

複雑な式に対しては，式を構成する各部分式の評価結果を知りたいことがあるかもしれません。

関数呼び出しを含む複雑な式，たとえば次のような式で，

```
if (isvalid(x) && !useless(x)) {
    ...
}
```

関数の戻り値を調べるには次のようにします。まず**F7**を押して呼び出される関数の先頭の行に入ります。そして関数の終わりのカッコ(})の上にカーソルを置いて**F4**を押し，その位置まで実行します。次に**F10**/Data/Function returnコマンドを選択してその関数から返される値を見ます。式中に他の関数呼び出しがあるときは，もう一度**F7**を押して次の関数の先頭の行に入り，同じようにしてその関数の戻り値を調べます。

関数呼び出しを含まない複雑な式，たとえば次のような式で，

```
if (x <= 5 && y[z] >8) {
    ...
}
```

各部分式の評価結果を見たいときは，CPUウィンドウを開いて1命令ずつ実行し，CPUレジスタに置かれている各部分式の値をウォッチします。

Turbo Assemblerのための助言

データの16進ダンプ

F10/Data/WatchコマンドまたはF10/Data/Evaluateコマンドと書式指定子の組み合わせにより，データをダンプすることができます。たとえば次は，

```
[ES:DI],20m
```

ES：DIレジスタ対の指し示すアドレスから20バイトのメモリ領域の内容を，そのまま16進数で表示するように指定します。

ソースレベルのデバッグ

アセンブラコードも，高級言語のプログラムと同じように，Moduleウィンドウを使って1行ずつ実行することができます。Moduleウィンドウの右側にRegisterウィンドウを置くと，レジスタの値を同時に見ることができます。

アセンブラ命令と機械語命令を同時に見たいことがあるかも知れません。その場合には，CPUウィンドウを開いて，CodeペインのMixedコマンドでBothを選択してください。このようにすると，アセンブラソースコードと機械語のコードバイトの両方を見ることができます。機械語のコードバイトを見たいときは，**F5**を押してCPUウィンドウをズームすることを忘れないでください。

レジスタの内容を調べる，変更する

レジスタの内容を変更する最も分かりやすい方法は，CPUウィンドウまたはRegisterウィンドウ内で，レジスタをハイライト表示させることです。もっと手早くレジスタの内容を変更することができるひとつの方法は，**F10**/Data/Evaluate/Modifyコマンドを使うことです。このコマンドを使うと，レジスタの内容を直接変更する代入式を入力することができます。たとえば次のように入力すると，SIレジスタに99がロードされます。

```
SI=99
```

同じ手法を使って，レジスタの内容を調べることもできます。たとえば

```
GRPH-D E AX
```

と入力すると，AXレジスタの内容が表示されます。

Turbo Pascalのための助言

初期化コードを1行ずつ実行する

最初にプログラムをTurbo Debuggerにロードしたときは，右向き矢印(→)は，プログラムのメインボディの**begin**キーワードを指しています。この**begin**は，実際には，プログラムが使用するすべてのユニットの初期化セクション(それらのユニットが初期化コードを持っているとき)に対する一連の呼び出しに対応します。すべてのプログラムは，***System***ユニットの初期化コードに対する呼び出しで始まります。

この時点で***F7***（Run/Trace intoコマンドのショートカット）を押すと，初期化コードを持つ最初のユニットに入ります。***F7***を使って最初のユニットの初期化コードの**end**を通り過ぎると，次のユニットに入ります。そのようにして最後には，プログラムのメインボディに戻り，最初の文を実行する準備が整います。

他方，プログラムの先頭で***F8***（Run/Step overコマンドのショートカット）を押すと，すべての初期化コードをスキップして，ただちにプログラムのメインボディを実行することができます。

終了手続きを1行ずつ実行する

プログラムが終了すると，一連の終了手続きに順番に制御が渡されます（Turbo Pascalリファレンスガイド第15章を参照してください)。プログラムのメインボディを1行ずつ実行して**end**を通過したときは，終了手続きの中に入りません。一連の終了手続きの中を通り抜けるには，デバッグしたい各手続きの出口にブレークポイントをセットしてください。

定数

定数の識別子は，スカラーと型つき定数だけしか識別されません。たとえば

```
program Test;
const
  A = 5;
  B = pi;
  Message = 'Testing';
  Caps = ['A'..'Z'];
  digits : string[10] = '0123456789';
begin
  Writeln(A);
  Writeln(B);
  Writeln(Message);
  Writeln('A' in Caps);
  Writeln(Digits);
end.
```

このプログラムでは，*A*(スカラー定数)と*Digits*(型つき定数)をインスペクトすることはできますが，*B*(浮動小数点数定数)，*Message*(文字列型定数)，*Caps*(集合型定数)はインスペクトできません。

文字列演算および集合演算の中間結果の一時格納スペース

CPUウィンドウを使っているときは，Turbo Pascalは，自動的に，文字列演算と集合演算の中間結果を一時的に格納するためのスペースをスタック上に確保します。

文字列に対して使用されているプラス(+)演算子と，すべての文字列関数は，演算の中間結果を格納するためのスタックスペースを確保します。このスタックスペースは，呼び出し側のスタックフレーム内に確保されます。同様に集合和(+)，集合差(−)，集合積(*)などの集合演算子も，中間結果のためにスタックスペースを確保します。

巧妙な型キャスト

Dosユニットには，あらかじめ定義されたすべてのファイル型の内部データフォーマットが定義されています。これらの宣言を使うと，すべてのファイル型変数のデータを調べることができます。次のプログラムを入力してみてください。

```
program Typecast;
uses Dos;
Var
  TextFile : Text;
  IntFile : file of Integer;
begin
  Assign(TextFile, 'TEXT.DTA');
  Rewrite(TextFile);
  Assign(IntFile, 'INT.DTA');
  Rewrite(IntFile);
  Close(TextFile);
  Close(IntFile);
end.
```

さらに次の4つのウォッチ式を追加してください。

```
IntFile
TextFile
FileRec(IntFile),r
TextRec(TextFile),r
```

最初の2つは，ファイルステータス（CLOSED, OPEN, INPUT, OUTPUT）とディスクファイル名を示します。また他の2つは，型キャストを使って，内部のフィールド名とファイル変数の値を表示します。

PascalでCPUウィンドウを使うときのアドバイス

■ ***System***ユニット内のルーチンには名前が付けられていません。したがってCPUウィンドウの中で**call**命令をウォッチすると，シンボル名のかわりに絶対アドレスが表示されます。

■ 複数のI/Oルーチン，たとえば***Readln***や***Writeln***は，しばしば複数のアセンブリ言語の**call**を生成します。

■ 範囲チェック，スタックチェック，I/Oチェックは，それらの機能を実行するライブラリルーチンの呼び出しを生成します。

■ 複数の演算子（longintの乗算，文字列の連結，その他）は，ライブラリルーチンの呼び出しを使って実行されます。

■ 手続きの直定数(文字列，集合，浮動小数点数)は，コードセグメント内の，その手続きのエントリポイントの直前に置かれます。

付録J

用語集

この付録には，このマニュアルでよく使用されている語が掲載されています。コンピュータやソフトウェアの分野で一般的に使用されている語と，Turbo Debuggerの環境において特別の意味で使用されている語があります。

ASCII

“American Standard Code for Information Interchange”（情報交換のための米国標準コード）の略。この標準規格で規定された文字および制御命令のセットのコード（符号）。

Cの式　Cexpression

C言語の文法にしたがって書かれた式。

CPU　Central Phocessing Unit

中央処理装置。このマニュアルでは，80x86プロセッサを意味します。CPUの内部には，多数のレジスタやフラグがあります。Turbo DebuggerのCPUウィンドウは，CPUの現在の状態を表示します。

CPUのフラグ　CPU flag

CPUがある状態に設定されていること，または特別の状態が発生したことを示すために使用されるビット。フラグレジスタの各ビットをいいます。

CPUレジスタ　CPU register

読み書きの時間を短縮するためにCPUチップの内部に設けられたデータの格納場所。80x86プロセッサには，AX, BX, CX, DX, SI, DI, BP, SP, CS, DE, ES, SSその他のレジスタがあります。

CS：IP

コードセグメント(CS)レジスタと命令ポインタ(IP)レジスタの対により指し示されるアドレス。このアドレスを，カレントプログラムロケーションと呼びます。

EMS Expanded Memory Specification

拡張メモリ仕様。Turbo Debuggerは，デバッグするプログラムのためにできるだけ広いメインメモリの領域を残すことができるように，プログラムのシンボルテーブルをEMSメモリに格納します（メインメモリに格納するように設定を変更することもできます）。

アクション action

ブレークポイントがトリガされたときに実行される処理や動作。アクションには，プログラムの実行の停止，式の値のログへの書き込み，式文の実行などがあります。

アクティブウィンドウ active window

画面上の現在ユーザが対話しているウィンドウ。ウィンドウが重なり合っているときは，一番上のウィンドウだけがアクティブウィンドウになることができます。アクティブウィンドウは，タイトルが反転表示され，枠線が2重になります。

アクティブペイン active pane

アクティブウィンドウの中の，現在ユーザの入力を受け付けているペイン。すべてのカーソル移動コマンドおよびローカルメニューコマンドは，このペインに作用します。

アセンブラ assembler

機械語命令を人間に読みやすくしたアセンブラ命令を機械語に翻訳するプログラム。

インスペクタ inspector

データの値を調べたり，変更したりするために使用されるウィンドウ。

ウィンドウ window

画面上で互いに独立に情報を表示することができる四角形の領域。**アクティブウィンドウ**の説明を参照してください。

ウォッチポイント watchpoint

式が真になるのを監視するブレークポイント。

演算子　operator

オペランドに対する演算を示す記号。たとえば，加算演算子(＋)や乗算演算子(＊)などです。

オペランド　operand

演算子や命令により演算や操作されるデータ。たとえば，3＊4では3と4がオペランドです。

逆アセンブラ　disassembler

機械語の命令をアセンブラ命令に変換するプログラム。CPUウィンドウのCodeペインには，自動的に逆アセンブルした命令が表示されます。

キャスト　casting

オペランドのデータ型を一時的に指定したデータ型に変換すること。型キャストまたは型変換ともいいます。

グローバルブレークポイント　global breakpoint

1ソース行または1命令実行されるごとに設定した条件がテストされ，条件が成立するときは設定したアクションが実行されるブレークポイント。

構造体　structure

Cのデータ型のひとつで，異なる型のデータをひとまとめにしたもの。Pascalのレコードに相当します。

コンフィギュレーションファイル　configuration file

Turbo Debuggerの画面やマクロの設定が入っているファイル。

式　expression

使用する言語の文法にしたがって書かれた，演算子とオペランドの組み合わせ。Turbo Debuggerがサポートする(式を評価できる)言語は，C，Pascal，アセンブラです。

自動変数　auto-variable

C言語で，関数が呼び出されて実行されるときにスタック上に生成され，実行が終了したときに消滅する変数。自動変数のスコープは，そのブロック内（Cでは一対のカッコ({ })によって囲まれたソース行）だけです。

シンボル　symbol

変数，定数，手続き，関数などの名前。

スカラー　scalar

Cのchar, integer, floatや，Pascalのbyte, integer, char, booleanのような基本データ型。スカラーは，配列や構造体のような，より複雑なデータ型の構成要素になることができます。

スコープ（通用範囲）　scope

ある変数が“見える”プログラムの範囲。グローバルなスコープを持つすなわちプログラムのどこからでもアクセスできる変数や，モジュールや手続きの内部でしかアクセスできないローカルな変数があります。

スタック　stack

関数や手続きの呼び出しに関係するデータ（リターンアドレス，呼び出しパラメータ，その他）が格納されるメモリの領域。

ステップ　step

関数や手続きの呼び出しを単一の要素として扱いながら，プログラムを1ソース行ずつ実行すること。1行ずつ実行したくないルーチンをスキップすることができます。

タイプ　type

データの型。各言語で使用できるデータ型については，その言語の説明書を参照してください。

デフォルト　default

ユーザが指定しなかったときに自動的に使用される値や設定。

トリガされる　triggered

設定されたすべての条件（パスカウント，マッチング条件など）が真になって，ブレークポイントのアクションが実行されること。

トレース　trace

プログラムを1ソース行または1命令ずつ実行すること。実行する行に関数や手続きの呼び出しがあるときは，その関数や手続きに入っていきます。

トレースポイント　tracepoint

変数またはメモリの領域が変化するのを監視するグローバルブレークポイント。

配列　array

CやPascalのデータ型のひとつ。型の同じデータをまとめたもの。配列の要素は，添字で指定することができます。

パス　path

階層構造のディレクトリにおけるルートディレクトリから目的のディレクトリまでの道筋。Turbo Debuggerが実行可能なプログラムを探すディレクトリは，DOSの環境変数PATHで指定することできます。

ヒストリリスト　history list

各プロンプトごとに保存された，そのプロンプトに対するユーザの応答のリスト。入力を要求するプロンプトが現われたときは，そのプロンプトのヒストリリストの中から選択することができます。

副作用　side effect

式が評価されたとき変数すなわちメモリロケーションの値を変更する作用。たとえば，代入演算子を含む式や，データの値を変更する関数を呼び出す式には副作用があります。

プルダウンメニュー　pull-down menu

メインメニューの項目を選択することにより開くメニュー。プルダウンメニューのコマンドは，どのウィンドウでも有効です。

ブレークポイント　breakpoint

プログラムの実行がその場所に到達したとき，設定されたアクションが実行されるプログラム内のアドレス。**アクション**の説明を参照してください。

プレフィクス　prefix

オペランドの前に置かれた演算子。たとえばCでは＋＋x。

索　引

16進数データ　420
16進表示
整数の～　74
ファイルの～　158,320
80386プロセッサ　14,352
仮想デバッグ　289-296
命令　293
レジスタ　169,195
80x87コプロセッサ　189,211
80x86プロセッサ　183
8087コプロセッサ　2,189,323
::(ダブルコロン)演算子　172,174
*(アスタリスク)
検索ワイルドカード　155,160,240
ブレークポイント
ウィンドウ内の～　136
,(カンマ)セパレータ　101
#(シャープ記号,
スコープオーバーライド)　165
.(ピリオド,
スコープオーバーライド)　165
?(疑問符)
検索ワイルドカード　155,160,240
;(セミコロン)
Cのプログラムの中の～　254
Pascalのプログラムの中の～　262
$R(Pascalコンパイラオプション)　264
-(ハイフン,
コマンドラインオプション)　66
-bオプション(TD386)　292
-doオプション　66
-dpオプション　67
-dsオプション　67
-eオプション(TD386)　290
-gオプション　67
-hオプション　67
-iオプション　68
-lオプション　68
-mオプション　68
-rオプション　68,371,372
-rpオプション　68
-rpオプション(TDRF)　304
-rsオプション　69
-rsオプション(TDRF)　304
-sオプション(TDSTRIP)　312
-scオプション　69
-sdオプション　69
-tオプション(TD386)　292
-v(TCCのオプション)　62
-vgオプション　70
-vnオプション　70
-wオプション(TDRF)　304
-wオプション　292
-zi(Turbo Assemblerのオプション)　63
-?オプション　67
/v(TLINKのオプション)　62,312
=, ==(等号と二重等号)Cの演算子　252
(下線記号,Cのシンボルの先頭)　169

【A】

Active user G-pageコマンド(Options/
Hardware statusメニュー)　76
ADD命令　189
Addローカルコマンド
(Breakpointsウィンドウ)　137,140
Add commentローカルコマンド
(Logウィンドウ)　141
Addressオプション
(Breakpointsウィンドウ/
Condition/Hardwareメニュー)　356

Always(Breakpoints/
Conditionローカルコマンド)　135
Animateコマンド(Runメニュー)　92
Anotherコマンド(Viewメニュー)　154
Argumentsコマンド
(Optionsメニュー)　75,98
ASCII 425
Dataペインの表示　198
～ファイル　157,316
Assembleローカルコマンド
(Codeペイン)　193
Atコマンド
(Breakpointsメニュー)　129,142
Atron　14

【B】

Beep on errorコマンド　342
Blockローカルコマンド
(Dataペイン)　202
Break(Set actionコマンド)　134
Breakpoint at(Get infoメッセージ)　88
Breakpointsウィンドウ　30,132
ローカルメニュー　133,226
Breakpointsメニュー　129,223
Byte(Dataペイン/
Display asローカルコマンド)　201
Buffer of symbolオプション
(TINST) 346

【C】

Cプログラミング言語 → Turbo Cも参照
==対=　252
breakキーワード　256
演算子　252
型キャスト　174
関数　3,113,173,255
桁の切り詰め　253
構造体と共用体　111
式　163
複雑な～　419
自動変数　251,255
シンボル　169
スカラ　108
セミコロン　254
タイプミス　256
定数　170,171
配列　111
バグ 251
副作用　102,173,254
符号の拡張　253
ポインタ　109,252,418
文字列　171
Callerローカルコマンド
(Codeペイン)　191
CALL命令　189,207
CD(TDRFのコマンド)　308
CGA　336
Changed memory globalコマンド
(Breakpointsメニュー)　130,144
Changed memory(Breakpoints/
Conditonローカルコマンド)
135,143,145
Changeローカルメニューコマンド
Registerペイン　194,215
Dataペイン　200
Globalペイン　83
Inspectorウィンドウ　52,59,124
Stackペイン　206
Staticペイン 84
Watchesウィンドウ　107

Clearコマンド(Dataペイン/
Blockローカルメニュー) 203
Close log fileローカルコマンド
(Logウィンドウ) 141
CodeView 63,300
CodeView→Turbo Debugger
(TDCONVRTユーティリティ) 300
Codeペイン 185,188
逆アセンブラ 188
ローカルメニュー 189,228
Comp(Dataペイン/
Display asローカルコマンド) 202
Complete graphics save
(TDINSTのコマンド) 339
COMファイル 313,314,317
Conditionローカルコマンド
(Breakpointsウィンドウ) 135
CONFIG.SYS 290
Control Break
(Get infoメッセージ) 88,89
Controlペイン
(Numeric Processorウィンドウ) 33
ローカルメニュー 217,232
COPY(TDRFのコマンド) 305
COPYFROM(TDRFのコマンド) 305
CPU 32,33
アセンブラプログラミングも参照
定義 425
フラグ 186,425
レジスタ 32,81,183,209,425
CPUウィンドウ 62,185
Codeペイン 188
逆アセンブラ 188,427
ローカルメニュー 189
Dataペイン 198
ローカルメニュー 198
Flagsペイン 196
ローカルメニュー 196
Registerペイン 194
ローカルメニュー 194
Stackペイン 204
ローカルメニュー 204
Pascalのためのアドバイス 424
ペイン 186,228
CPUコマンド(Viewメニュー) 108,185
Createコマンド(Macrosメニュー) 73,97
CS:IP 81,86,87,156,179,186,426
Ctrl-Break(割り込みキー) 94
CTRL-STOP(割り込みキー) 94
Cursor switchコマンド(Options/
Hardware statusメニュー) 77
CVPACK 301
CXレジスタ 268
Cycle typeオプション
(Breakpointsウィンドウ/
Condition/Hardwareメニュー) 355

【D】

-doオプション 66
-dpオプション 67
-dsオプション 67
data 99
型変換 107
~のインスペクト 100
~のウォッチ 104
~の変更 101
Dataオプション(Breakpointsウィンドウ
/Condition/Hardwareメニュー) 356
Dataペイン 198
ローカルメニュー 198,229

Dataメニュー 100,223
Decrementローカルコマンド
(Registerペイン) 195
DEL(TDRFのコマンド) 306
Delete allコマンド
Breakpointsメニュー 130
Macrosメニュー 73
Delete allローカルメニューコマンド
Breakpointsウィンドウ 138
Watchesウィンドウ 107
Descendローカルコマンド
(Inspectorウィンドウ) 124
DIR(TDRFのコマンド) 306
Display asローカルコマンド
CPUウィンドウ/Dataペイン 200
Fileウィンドウ 161
Display swappingコマンド
(Environmentメニュー) 74
DOS 323
～に戻る 266
～の呼び出し 78
Double(Dataペイン/
Display asローカルコマンド) 202
Dump(Viewメニューコマンド) 209
Dumpウィンドウ 32,209

【E】

-eオプション 291
Editローカルメニューコマンド
Fileウィンドウ 21,161
Moduleウィンドウ 156
Watchesウィンドウ 106
EGA 70,336,338
EMS 62,88,326,344,369,426
EMS/VRAM switchコマンド(Options/
Hardware statusメニュー) 77
Enable/Disableローカルコマンド
(Breakpointsウィンドウ) 137
Environmentコマンド
(Optionsメニュー) 73
Erase logローカルコマンド
(Logウィンドウ) 141
ESCキー 34,391
Evaluate/modifyコマンド
(Dataメニュー) 101,163,420
Executeコマンド(Breakpointsウィンドウ/
Set actionメニュー) 134
Execute toコマンド
(Runメニュー) 46,55,92
EXEファイル 2,65,98,312
～の圧縮 317
～のダンプ 318
Expression true(Breakpoints/
Conditionローカルコマンド) 136
Expression true global(Breakpoints
メニューコマンド) 130,144
Extended(Dataペイン/Display as
ローカルコマンド) 202

【F】

FAR命令 207
Fast screen updateコマンド 338
Fileローカルコマンド
Fileウィンドウ 161
Moduleウィンドウ 154
Fileウィンドウ 31,157,230
ローカルメニュー 158
Fileメニュー 222
Flagsペイン 186
ローカルメニュー 196,230

Float(Dataペイン/Display as
ローカルコマンド) 202
Followローカルメニューコマンド
Codeペイン 191
Dataペイン 200
Stackペイン 206

【G】

Get infoコマンド 87
Globalローカルコマンド
(Breakpointsウィンドウ) 137,144
Globalペイン 82
ローカルコマンド 82
Global Symbolペイン
ローカルメニュー 233
Go to cursorコマンド
(Runメニュー) 46,54,91
Gotoローカルメニューコマンド
Codeペイン 190
Dataペイン 199
Fileウィンドウ 159
Moduleウィンドウ 156
Stackペイン 205
Graphics switchコマンド(Options/
Hardware statusメニュー) 76
Graphics swtchesオプション
(TINST) 346
GVRAM 67,76,346

【H】

-hオプション 67
Hardware(Breakpointsウィンドウ/
Conditionローカルコマンド) 136
Hardwareメニュー 355
Hardware statusコマンド
(Optionsメニュー) 76
Helpウィンドウ 38

【I】

-iオプション 68
I/Oローカルコマンド(Codeペイン) 193
IDの切り替え 68,323,344
In byte(Codeペイン/
I/Oローカルコマンド) 194
Incrementローカルコマンド
(Registerペイン) 195
Inspectorウィンドウ 34,108,426
Cの～ 108,109,111,113
Pascalの～ 114-118
アセンブラの～ 118,119,121,122
関数の～ 113,117
構造体と共用体の～ 111,121
スカラーの～ 108,114,118
～のローカルコマンド 123
配列の～ 111,116,121
ポインタの～ 109,115,119
レコードの～ 117
Inspectコマンド 25
Breakpointsウィンドウ
ローカルメニュー 138
Dataメニュー 49,57,100
Globalペインメニュー 82
Inspectorウィンドウ
ローカルメニュー 124
Moduleウィンドウ
ローカルメニュー 103,153
Stackウィンドウ
ローカルメニュー 85
Staticペインメニュー 83
Watchesウィンドウ

ローカルメニュー　107
Instruction traceコマンド
(Runメニュー)　93
Integer formatコマンド
(Environmentメニュー)　74
Interrupt(Get infoメッセージ)　90
IPTR → 命令ポインタ参照

【J】

JMP命令　189,207

【K】

Keysオプション　341

【L】

-lオプション　68
Languageコマンド
(Optionsメニュー)　72,164
LIBファイル　318
Lineローカルコマンド
(Moduleウィンドウ)　155
Listペイン　226,237
Loaded(Get infoメッセージ)　90
Loadコマンド　96
Local symbolペイン
ローカルメニュー　233
Localsローカルコマンド
(Stackウィンドウ)　86
Loggingローカルコマンド
(Logウィンドウ)　141
Logウィンドウ　31,139,231
ローカルメニュー　140
Logコマンド
Breakpoints/Set action
ローカルメニュー　146
Set actionローカルメニュー　134
Logファイル　141
～の長さ　338
変数　146
Long(Dataペイン/
Display asローカルコマンド)　201
Long followローカルコマンド
(Dataペイン)　200

【M】

-mオプション　68
Macrosメニュー　72
MAPファイル　316
MASM　272,327
Maximum tiled watchオプション　338
MD(TDRFのコマンド)　307
Microsoft 10,20,63,299
Microsoft Macro Assembler　272
Mixiedローカルコマンド
(Codeペイン)　188,192,420
Mode for displayメニュー
(TINST)　347
Mode for hardwareメニュー
(TINST)　345
Moduleローカルコマンド
(Moduleウィンドウ)　154
Moduleウィンドウ　30,150,231
ローカルメニュー　153
Move/Resizeコマンド　36
Move(Dataペイン/
Blockローカルコマンド)　203
MOV命令　189
MS-LINK　302

【N】

NEAR命令 207
New CS:IPローカルコマンド
(Codeペイン) 192
New expressionローカルコマンド
(Inspectorウィンドウ) 125
Next pane cycleコマンド 35
Nextローカルメニューコマンド
Dataペイン 199
Fileウィンドウ 161
Moduleウィンドウ 154
No Program Loaded
(Get infoメッセージ) 88
Numeric processorコマンド
(Viewメニュー) 213
Numeric Processorウィンドウ
33,213,232

【O】

OBJファイル 320
OFFSET演算子 208
Open log fileローカルコマンド
(Logウィンドウ) 141
Optionsメニュー 72,224
Originローカルメニューコマンド
Codeペイン 190
Moduleウィンドウ 87,156,167
Stackペイン 205
OS shellコマンド 78,293
Out byte(Codeペイン/
I/Oローカルコマンド) 194

【P】

Pascal → Turbo Pascalも参照
CPUウィンドウ 424
演算子 177
型キャスト 423
関数 3,117,262
呼び出し 178
境界エラー 264
空文 262
サブプログラム 3
式 163,176
呼び出しルーチン 178
初期化コード 422
終了手続き 422
シンボル 176
スカラー 114
スコープ 260
スタック 423
定数 176,422
手続き 3,178
ネストされたルーチン 260
配列 116
範囲チェック 264
ポインタ 115,259
文字列 177
文字列および集合の
一時格納領域 423
ユニット 3
ユニットのオーバーライド 167
レコード 117
Pass countローカルコマンド
(Breakpointsウィンドウ) 136,143
PATH(DOSコマンド) 429
Path for sourceコマンド
(Optionsメニュー) 75,151
Periscope 14
Previousローカルメニューコマンド

Codeペイン　191
Dataペイン　200
Moduleウィンドウ　155
Stackペイン　206
PROC疑似命令　266
Program resetコマンド
(Runメニュー)　91,95
Prompting(TDINSTのコマンド) 342

【Q】

Quitコマンド　78

【R】

-rオプション　68,372
$R(Pascalのコンパイラ指令)　264
Rangeローカルコマンド
(Inspectウィンドウ)　115,123
RD(TDRFのコマンド)　307
Read word(Codeペイン/
I/Oローカルコマンド)　194
Read(Dataペイン/
Blockローカルコマンド)　204
README.COM　11
Real(Dataペイン/
Display asローカルコマンド)　202
Registers 32-bitローカルコマンド
(Registerペイン)　196
Registersウィンドウ　32,209
Registerペイン
CPUウィンドウ　194,228
ローカルメニュー　194,228,230
Numeric Processorウィンドウ 214
ローカルメニュー　215,232
Remote File transfer 303
→ TDRFユーティリティ
Removeコマンド
Macrosメニュー　72
Breakpointsウィンドウ
ローカルメニュー　138
Watchesウィンドウ
ローカルメニュー　107
REN(TDRFコマンド)　307
Restore optionsコマンド
(Optionsメニュー)　76
RETF命令 189
RET命令 189,266
-rp(TDRFのオプション)　304
-rpオプション　68
-rs(TDRFのオプション)　304
-rsオプション　69
Runコマンド(Runメニュー)　90
Runメニュー　90,223

【S】

-sオプション(TDSTRIP)　312
Save optionsコマンド
(Optionsメニュー)　36,75,97
-scオプション 69
Screen modeオプション(TINST) 345
Screen sizeコマンド
(Environmentメニュー) 74
-sdオプション 69
Searchローカルメニューコマンド
Codeペイン 191
Dataペイン 199
Fileウィンドウ 159
Moduleウィンドウ 28,155
Set actionローカルコマンド
(Breakpointsウィンドウ) 133

Set(Dataペイン/
Blockローカルコマンド 203
Source directoriesコマンド
(TDINST) 241
Source(Languageメニューコマンド) 164
SS:BP 210
SS:SP 186,205
STACK疑似命令 267
Stackウィンドウ 30,85,233
Stackペインローカルメニュー 204,229
Staticペイン 82
ローカルコマンド 83
Statusペイン(Numeric
Processorウィンドウ) 214
ローカルメニュー 216,232
Step(Get infoメッセージ) 90
Step over(Runメニューコマンド) 91
step(ステップ) 20,428
Stop recordingコマンド
(Macrosメニュー) 73
Stopped at(Get infoメッセージ) 88

【T】

Tab column widthオプション
(TDINST) 338
Tab sizeコマンド
(Environmentメニュー) 75
Turbo Assembler 2,63,327
アセンブラプログラミングも参照
Quirksモード 272
Turbo Debuggerの組み込み
アセンブラとの比較 207
デバッグの準備 63
Turbo C 2,62,169,251 C言語も参照
コードの最適化 418
コードの生成 210
デバッグの準備 62
リンク 272
TCDEMO.C 42
TD.EXE 12
TD386.EXE 291
TDCONVRT.EXE 63,300
TDH386.SYS 14,136,290,296,352
TDHELP.TDH 11
TDINST.EXE 331
TDMAP.EXE 316
TDPACK.EXE 317
TDREMOTE 303,369
起動 372
メッセージ 375
TDREMOTE.EXE 371
TDRFユーティリティ 304
CDコマンド 308
COPYFROMコマンド 305
COPYコマンド 305
DELコマンド 306
DIRコマンド 306
MDコマンド 307
RDコマンド 307
RENコマンド 307
コマンドラインオプション 304
メッセージ 308
TDSTRIPユーティリティ 312
エラーメッセージ 313
オプション 312
Terminated(Get infoメッセージ) 89
Textペイン 235
TLINK 62,312
Toggleコマンド
Breakpointsメニュー 129,142

Flagsペイン
ローカルメニュー 196
Controlペイン
ローカルメニュー 217
Statusペイン
ローカルメニュー 216
TPC.EXE 281
TPDEMO.PAS 42
Trace intoコマンド
(Runメニュー) 47,54,90
Trace(Get infoメッセージ) 88
Turbo directoryオプション 341
Turbo Pascal 2,62,81,264
→Pascalも参照
コンパイル 281
初期化されない変数 259
デバッグセッションの例 281
デバッグの準備 62,281
TURBO.EXE 281

【U】

Undo closeコマンド 44
Until returnコマンド
(Runメニュー) 47,54,92
User screenウィンドウ 33,375

【V】

-vオプション(TCC) 62
/vオプション(TLINK) 62,312
-vgオプション 70
Visual user G-pageコマンド(Options/
Hardware statusメニュー) 76
VGA 70,336,338
Viewメニュー 222
-vnオプション 70
void関数 3

【W】

-w(TDRFのオプション) 304
-wオプション 292
Watchesウィンドウ 30,48,56,104,233
値の変更 106
インスペクト 107
式の削除 107
~のカスタマイズ 338
編集 106
Watchコマンド
Dataメニュー 102
Moduleウィンドウ
ローカルメニュー 154
Watchesウィンドウ
ローカルメニュー 106
Windowメニュー 224
Word(Dataペイン/
Display asローカルコマンド) 201
WordStar 237,319,342
Write word(Codeペイン/
I/Oローカルコマンド) 194
Write(Dataペイン/
Blockローカルコマンド) 204

【Z】

Zeroローカルメニューコマンド
CPUウィンドウ/
Registerペイン 195
Numeric Processorウィンドウ/
Registerペイン 216
-zi(Turbo Assemblerオプション) 63

【あ】

アクション(定義)　426
アクティビティインジケータ　37
アクティブウィンドウ(定義)　426
アクティブペイン(定義)　426
アスタリスク(*)
　検索ワイルドカード　155,160,240
　ブレークポイント
　　ウィンドウ内の～　136
アセンブラ(組み込み)　207
アセンブラプログラミング　118-122
　→ CPU,Turbo Assemblerも参照
　16進数データ 420
　ADD 189
　CALL 189,207
　CX レジスタ　268
　DOS　266
　FAR　207
　JMP 189,207
　MOV 189
　NEAR 207
　OFFSET 208
　PROC 266
　RET　189,266
　RETF 189
　インライン命令　327
　演算子　180
　オペランド　207,266,270
　サブルーチン　266
　式　163,179
　ジャンプ命令　267
　乗算　270
　シンボル　179
　スタックスペース　267
　ストリング命令　268
　ストリング命令の
　　デフォルトセグメント　269
　セグメントグループ　272
　セグメントの一巡
　　(ラップアラウンド)　271
　ソースレベルのデバッグ　420
　定数　179
　ディレクションフラグ　268
　バイトのワードへの変換　269
　複数のプレフィクス　270
　フラグ　270
　メモリ参照オペランド　271
　文字列　208
　文字列の比較　269
　レジスタ　270, 420
　割り込みハンドラ　271
アドレス　205
インストール(Turbo Debuggerの)　11
インストレーション　11,331-349
　～の保存　349
インスペクト　20
ウィンドウ　44,426
　Breakpoints　30
　CPU　32
　Dump　32
　File　31
　Help　37
　Inspector　34,50,57
　Log　31
　Module　30
　Numeric Processor　33
　Registers　32
　Stack　30,85
　User Screen　33

Variables　31,80
Viewメニューから
オープンできる～　30
Watches　30
アクティブ～　34,45,426
～操作コマンド　239
～の再設定　36,45
～のズーム　36
～のペイン　35,426,430
List　237
Text　35,237
～の保存　36
～を順番にアクティブにする　35
～を複数個オープンする　34
ウォッチする　21
ウォッチポイント　127,426
→ ブレークポイントも参照
エディタのカスタマイズ 340
エミュレーション　212
エラーメッセージ　381
TDCONVRT　300
TDH386.SYS　296
TDREMOTE　375
TDRF　308
TDSTRIP　313
仮想デバッガ(TD386)　294
致命的～　391
～の消去　342
ビープ　390
メモリ不足の～　369
演算子
Cの～　172,252
Pascalの～　177
アセンブラの～　180
定義　426
～の即値オペランド　189
～のプレフィクスとポストフィクス
426,429,436
大文字対小文字　69,344
オブジェクトファイル　320
オプション
→ コマンドラインオプション参照
オペランド
～の順序　266
定義　427

【か】

拡張メモリ仕様 → EMSを参照
カスタマイズ
→ インストレーションも参照
OSシェル　343
カラー　332
キー　341
言語　343
整数の表示書式　337
表示　335
表示行数　338
リモートデバッグ　342
ログファイル　338
下線記号"_"(Cのシンボル中の)　169
仮想デバッグ　289-296,322
型キャスト → 型変換参照
型変換　107,124,427
Cの～　174
Pascalの～　423
画面
行数　338
スノーノイズ　338
カラー(表示色)
カスタマイズ　332

スノー　338
関数
Cの～　141
～のインスペクト　113,117
定義　3
カンマ(,)セパレータ　101
キーストロークの記録 96
キー入力による選択　36
疑似変数　169
起動(Turbo Debugger)　61
起動時の表示のカスタマイズ　335
疑問符(?,検索ワイルドカード)
155,160,240
逆アセンブラ
→ CPUウィンドウのCodeペイン参照
キャスト → 型変換参照
行番号(スコープオーバーライド)　166
共用体　111,121
クイックリファレンス行 38
クラッシュ(システムの) 322
グラフィックスイメージ 70
グラフィックスディスプレイバッファ
339
グローバルメニュー 25
検索
キー入力による～　36
ワイルドカード　155,160,240
更新(ユーザ画面の)　336
構造体　427
構造体と共用体のインスペクト
111,121
コプロセッサ 2,189,211,323
コマンドブロック　361
コマンドラインオプション
62,63,297,244,331
-?　67
-b　292
-do　66
-dp　67
-ds　67
-e　291
-g　67
-h　67
-i　68
-l　68
-m　68
-r　68,372
-rp　68
-rs　69
-sc　69
-sd　69
-t　292
-vg　70
-vn　70
-w　292
TD386.EXE　293
TDREMOTE　373
TDRFユーティリティ　303
シンボル　69
対応するTDINSTのコマンド　348
ビデオハードウェア　70
表示の更新　67
ヘルプ　67
小文字対大文字　69,344
コンパイラ疑似命令　150
コンフィギュレーション
→ インストレーション参照
コンフィギュレーションファイル　427

【さ】

最下行　38
再帰　81,86
再設定(ウィンドウの)　36,45
最適化(コードの)　418
再ロード(プログラムの)　43,96
式　163
　Cの～の構文規則　169,425
　Pascalの～の構文規則　176
　アセンブラの～　176
　定義　169,425
　～のインスペクト　124
　～の評価　164
　　スコープ　167
　副作用　173
システムの破壊　322
自動変数(C言語)　251,255,427
シャープ記号#
　(スコープオーバーライド)　165
ジャンプ命令(アセンブリ言語)　267
終了(プログラム)　95
終了(Turbo Debugger)　44,78
16進数データ　420
16進表示
　整数の～　74
　ファイルの～　158,319
出力画面(プログラム)　33
ショートカット → ホットキー参照
状況に応じたヘルプ　37
情報メッセージ　415
初期化コード　422
書式制御文字列　101,181
シンボル　変数も参照
　Cの～　169
　Pascalの～　176
　アセンブラの～　179
　コマンドラインオプション　66
　定義　428
　～の逆アセンブル　188
　～のスコープ　165
　～へのアクセス　165
シンボルテーブル変換ユーティリティ　300
ズーム(ウィンドウの)　36
スイッチ
　→ コマンドラインオプション参照
スカラー
　定義　428
　～のインスペクト　108,114,118
　バイトリスト　168
スコープ　104,131
　暗黙の～　167
　シンボルの～　165
　定義　428
　～のオーバーライド　165,176
　～を間違える　260
スタック
　Pascalの～　423
　定義　428
スタックスペースの確保　267
ステータスブロック　365
ステップ(step)　16,428
スノーノイズの多い画面　338
整数の表示　337
セミコロン(;)
　Cプログラム中の～　254
　余分な～　262
ソースファイル　2,65
　～の拡張子　164
　～を調べる　150

ソフトウェアに必要な条件　2,64

【た】

ダブルコロン(::)演算子　172,174
致命的エラー　391
チュートリアル　41
中央処理装置　CPU参照
データ ポインタ　213
データ型　428 → 配列，ポインタ，
　レコード，スカラー，
　構造体，共用体も参照
定数
　Cの～　170
　Pascalの～　176,422
　アセンブラの～　179
ディレクトリ　13,303
　コンフィギュレーションファイル
　　69
　ソースファイル　69,75,151,341
　ヘルプファイルとコンフィギュ
　　レーションファイル　341
デバイスドライバ　351
デバイスドライバインターフェース
　　358
　仮想デバッガ　290
　ファンクションコール　359
　コマンドブロック　360
　ステータスブロック　365
デバッグ
　80386プロセッサ　289-296
　Microsoftのプログラム　20
　Pascalのポインタ　259
　Pascalの空文　262
　アセンブラレベル　183
　アセンブラに特有のバグ　265
　大きなプログラム　369
　隠れた副作用　257
　仮想　289-296,322
　　～対通常　293
　関数　262
　境界および限界のテスト　264,273
　サンプルセッション　275-288
　初期化されていないデータ　257
　初期化されていない変数　259
　シンボルテーブル　312
　スコープの間違い　260
　定義　17
　何も入力しないでテストする　273
　ネストされたルーチン　260
　～の手法　248,274
　～の準備　62,63
　～のスタイル　249
　～のステップ　18
　～の方針　277,282
　バグの種類　251
　範囲チェック　264
　ヒープスペースの不足　258
　フェンスポストエラー　258
　複数のバグ　277
　プログラムのテスト　273
　プログラムのトレース　20
　間違ったデータによるテスト　273
　リモート ～　369,375
　　オプション　68
　　カスタマイズ　342
　割り込み　324
電卓　101,163
トレース　20,46,54,428
トレースポイント　127,429
　→ ブレークポイントも参照

【な】

ニーモニック → 命令参照
ヌルモデムケーブル　370

【は】

ハードウェアデバッガ　351
ハードウェアに必要な条件　2
バイトリスト　168,192,199
ハイフン
　(コマンドラインオプション)　66
配列　429
　～のインスペクト　51,58,111,116,121
　部分範囲　123
バグ → デバッグ参照
パラメータ(定義)　3
範囲チェック　264
範囲エラー　264
ビープ　391
ヒープサイズ　68
引数(定義)　3
ヒストリリスト　27,342,382,429
ビデオディスプレイ　70
ビデオハードウェア　70
評価(式の)　164
表示
　更新　67
　～の切り替え(スワップ)　336
　パラメータ　335
　モード　325
表示形式(整数の)　74
表示の切り替え　336，
ピリオド
　(.,スコープオーバーライド) 165
ファイル　31
　16進表示　158,160,319
　ASCII～　158,160,319
　COM～　313,314,317
　EXE～　2,65,98,312
　　～のダンプ　319
　　～のデバッグ情報の圧縮　317
　LIB～　319
　MAP～　316
　OBJ～　320
　TDS～　313,317
　ソース ～　150
　　拡張子　164
　～のインスペクト　154
　～のウォッチ　153
　～の検索　155,159
　～のダンプ　318
　～の編集　156,161
　表示～の切り替え　154,160
　～メンテナンス操作　303
　～リスト　241
　～を調べる　149-161
ファンクション(デバイスドライバインターフェース)　359
副作用　102,147,173,254
　定義　429
浮動小数点数　211
浮動小数点レジスタ　214
フラグ　214,270
プルダウンメニュー 25,44,429
ブレークポイント 30,127-147
　グローバル～ 137,144,427
　条件～ 143
　スコープ 131
　～のインスペクト 138

～の解除 130,138
～のスピードアップ 351
～の設定 47,55,129,142
～の追加 137
定義 127,429
～のトリガ 428
ハードウェア～ 145,353,361
パスカウント 143
～を無効にする 137
プログラム
カレントロケーション 185
現在の状態 80
スタックポインタ 185
停止の原因 88
～の再ロード 43,96
マクロの使用 97
～の実行 46,54
～の実行の制御 79
～の実行の中止 94
～の終了 95
～の設計 274
～のデータの変更 99
～のテスト 273
～のデバッグのための準備 62
～のパッチ 180
～の変更 207
～のメモリへのロード 322
プログラムの出力 33
プロセッサ → 80386プロセッサ,
80x86プロセッサ,CPU参照
プロンプトボックス 27,238,381
ペイン 35,426,430
ベクター 325
ヘルプ 37,44
～行 38,44
コマンドラインオプションの～ 67
状況に応じた～ 38
～ファイルを探すディレクトリ 341
変数 30,80 → シンボルも参照
Cの自動～ 251,255,427
初期化されない～ 259
～の値のログ 146
～のインスペクト 49,57
～のウォッチ 21,102
～の指定 103
～のデクリメント 263
～の変更 21,52,59,83,99
Variablesウィンドウ 31,80,233
ポインタ
Cの～ 252
Pascalの～ 259
～のインスペクト 109,115,119
保存
ウィンドウ 36
画面の設定 36
ホットキー 24,220
～の有効/無効の切り替え 341
ポップアップメニュー 26,226,430

【ま】

マクロ 29,97
～の保存 75
マスターディスク 11
バックアップ 10
命令
80386 293
ADD 189
CALL 189,207
FAR 207
JMP 189,207

MOV　189
NEAR　207
RET　189,266
RETF　189
1命令ずつの実行　91
インライン　～　327
オペランド　208
～の表示　192
～のフェッチ　355
文字列～　208
命令のオペコード　213
命令ポインタ　213
メッセージ → エラーメッセージ参照
メニュー
プルダウン　25,44,429
プルダウン対ポップアップ　25
ポップアップ　25,226,430
メニュー構成　242
メニューバー　430
ローカル　25,26,226,430
メモリ
EMS～　62,88,326,344,426
アセンブラ　207
～アドレス　186
エラーメッセージ(メモリ不足)　369
仮想デバッグと～　290
上位～　289
～の確保　291
～の節約　339
～の使い方　325
～の内容の表示　32
ブレークポイント　130,135,144,145
～へのプログラムのロード　322
～変数　271
文字の大小の区別　344
モジュール(定義)　3
文字列
Cの～　171
Pascalの～　171
バイトリストの中の～　168
戻る(DOSに)　43

【や】

ユーザインタフェース　22
ウィンドウ　29-36
グローバルメニュー　25
状況感知能力　25
ヒストリリスト　27
ヘルプの呼び出し　37-39
マクロ　29
ローカルメニュー　26
ユーザ画面の更新　336
ユーティリティ
TDCONVRT　300
TDMAP　316
TDPACK　317
TDRF　303,375
TDSTRIP　312
TDUMP　318
ユニット(Pascalの)　3,422
用語集　425-430

【ら】

ライブラリファイル　320
リスト項目の選択　35
リターンキー　35
リファレンス行　38
リモートシステムとリモートファイル　303
リモートデバッグ　269,375

例題プログラム　42,275
レコード　117
レジスタ　169
　16ビット～対32ビット～　196
　CS～　188
　DS～　198
　I/O～　193
　関数の戻り値と～　210
　乗算と～　270
　～の退避　267
　～の変更　270,420
　浮動小数点～　214
ローカルシステム　303
ローカルファイル　303
ローカルメニュー　25,26,226,430

【わ】

ワイルドカード　430
割り込み　324
　～ベクター　324
割り込みキー　94,341

Turbo Debugger 1.0 ユーザーズガイド

1989年 4月 1日　初版発行

著者　**Borland International**

編訳・発行　**(株)ボーランド ジャパン**

〒107 東京都港区南青山7-8-1 小田急南青山ビル

電話 (03)486-1400(代表)

製品に関するお問い合わせは,

下記代理店までお願い致します。

(株)マイクロソフトウェア アソシエイツ

〒107 東京都港区南青山7-8-1 小田急南青山ビル

電話 (03)486-1411(代表)

(03)486-1403(サポートセンター)

FAX (03)486-8905

Printed in Japan

落丁,乱丁はお取り替えいたします。

TURBO DEBUGGER®
BORLAND
Copyright © 1988 Borland International, Inc. All rights reserved. Licensed material—program property of Borland International, Inc. 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001, A *Turbo Language*™ product. Unauthorized use, duplication, or distribution is strictly prohibited by federal law.
PART# 15LB-ASD03 BOR 0850
DISK 2
TURBO ASSEMBLER & DEBUGGER®
BORLAND

TURBO DEBUGGER®

USER'S GUIDE

BORLAND